TIDE: Task-Driven DNN Training and Splitting for Efficient Inference at the Mobile Edge

TIDE: Task-driven DNN Training and Splitting for Efficient Inference at the Mobile Edge

Francesco Malandrino[1,2], Gustavo De Veciana[3], Carla Fabiana Chiasserini[4,1,2]
1: CNR-IEIIT, Italy – 2: CNIT, Italy – 3: UT Austin, USA – 4: Politecnico di Torino, Italy

***Abstract*—The growing demands of DNN-based inference at the mobile edge is driving the need for increasingly efficient execution. Such applications often require fast and high-quality outputs, which are hard to realize due to the limited computational and communication capabilities at the edge. This paper tackles these issues focusing on a DNN for the execution of tasks that are homogeneous in nature but heterogeneous in their domains. The key idea is to start with a parent DNN of interconnected computational elements (atoms), and strategically form a collection of task-specific DNNs suitable for distributed deployment. Such task-specific DNNs may include common as well as uniquely used atoms of the parent DNN. Ultimately, the aim is that they be smaller in size – thus a better match for edge resources – and achieve low-cost inference. We solve the problem of determining the best collection of task-specific DNNs through an algorithmic framework named TIDE. Experimental results show that TIDE decreases inference cost and time by 90% and 80% (resp.) relatively to centralized approaches, and by over 60% and 70% (resp.) when compared to the best benchmark.**

***Index Terms*—Distributed inference, resource allocation, edge computing**

I. Introduction and Motivation

An increasing number of Machine Learning (ML) models are running at the communication network edge and in a distributed fashion [1], [2]. This poses various challenges as ML models become increasingly complex, while heterogeneous edge nodes (equipped with CPU or GPUs) may have different, and often limited, capabilities. Moreover, the latency of the network sections interconnecting nodes may impair the distributed execution of ML models. Due to these factors and the need for scalability, the joint optimization of the performance of ML-based tasks inference and their resource allocation has become a problem of paramount importance.

Problem and research gaps. This paper addresses the above issues in the relevant setting where Deep Neural Networks (DNNs) are designed to perform inference on a set of tasks that are homogeneous in nature (e.g., image classification) but heterogeneous in their domains (e.g., dealing with the classification of different objects, or with heterogeneous distributions of input samples). Many of these are needed in critical applications that require high-quality and fast inference; examples include object detection [3], sensor fusion [4], sentiment analysis [5], and medical diagnosis [6]. To address the quality issue, DNNs have to be specialized (i.e., fine tuned) for the different tasks and adapted as new tasks (or shifts in the input sample distributions) emerge.

To cope with that, various approaches have been proposed in the literature, mainly focusing on the design of adequate DNN architectures and their training. Overall, although effective in coping with learning specialization for multiple tasks, none of the above solutions is concerned with either the efficient and swift execution of tasks inference, or the distributed deployment of the resultant DNNs in edge environments.

Our solution. We address the above gaps as follows. *First,* we design a solution framework which takes an initial DNN architecture (referred to as *parent DNN*) as a starting point. The parent DNN is partitioned into a set of interconnected computational atoms (i.e., indivisible computational units such as convolutional filters).

Second, task-specific DNNs are derived from the parent DNN. Such specialization may include common as well as uniquely used portions of the parent DNN, as needed. The idea is indeed that such DNN specializations emerge in a dynamic way recognizing commonalities, hence synergies, across tasks as well as their distinctiveness. This not only ensures flexibility in shaping DNNs for high-quality task-specific inference, but also – and crucially – *low-cost inference execution with savings in computational and energy resources*.

Third, the proposed process for creating task-specific DNNs is designed to make them suitable for distributed deployment, hence, inference, on possibly heterogeneous physical resources. This facilitates inference execution of DNNs at the edge, as it allows a better matching between computational loads to edge resources.

We tackle these issues through an algorithmic framework named TIDE (for Task-driven DNN Training and Splitting for Efficient Edge Inference). TIDE's goal is to enable inference *(i) with high quality and low delay*, and that is *(ii) efficient*, and *(iii) distributed* whenever necessary or convenient.

Our contributions. We provide the following contributions:

• We design a methodology to derive collection of DNN architectures that support a set of homogeneous tasks with heterogeneous input distributions. To that end, we model an initial parent DNN as a set of interconnected computational atoms. This provides an avenue to dynamically activate an efficient set of DNN atoms that are relevant to the inference task at hand. We then use this model to formulate the problem of creating task-specific DNNs suitable for distributed deployment, to achieve low-cost inference (Sec. II).

• We introduce TIDE, an algorithmic framework that solves our problem through an efficient and flexible matching procedure to obtain high-quality choices of the computational atoms to be used by each task (Sec. III).

• Results show that TIDE's decreases the inference time and cost by 90% and 80% (resp.) relatively to centralized approaches, and by over 60% and 70% (resp.) relatively to the best benchmark (Sec. IV).

II. System Model and Problem Formulation

Our aim is to design a set of DNN architectures which are derived from the *same* parent DNN architecture and specialized to support a homogeneous set of *tasks* $\mathcal{T}$, i.e., tasks that have the same input and output dimensions, but possibly associated with heterogeneous distributions of input samples. The *specialization* of the parent architecture to tasks allows for more efficient inference cost for the given task, while the *shared* parent DNN architecture allows for dynamic adaptation as new tasks to handle arrive and the possibility of saving memory when task-specific DNNs that share atoms are deployed. We explain the system model below.

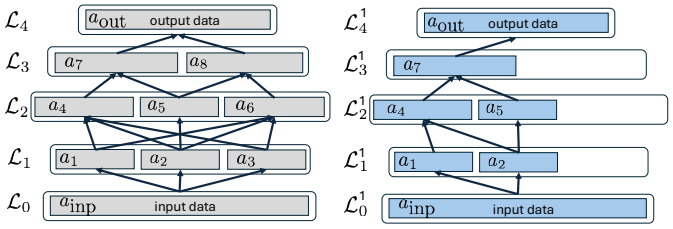

Fig. 1. Left: an example of parent DNN architecture highlighting the input data atom, the computational atoms (including the output data atom), the layers they belong to, and their connectivity. Right: an example of a task-specific DNN architecture derived for a given task.

Parent DNN architecture. The parent architecture consists of a set of atoms, $\mathcal{A}$, which includes: an *input* data atom, denoted by a_{inp}, and a collection of computational atoms including an *output* data atom, denoted by a_{out}. As shown in Fig. 1 (left), the atoms are partitioned into $L{+}1$ layers corresponding to subsets $\mathcal{L}_0, ...\mathcal{L}_L$ of $\mathcal{A}$; in our example, $L{=}4$. The atoms in a layer are connected to atoms in the next layer via a set of directed edges $\mathcal{E}{\subset}\mathcal{A}{\times}\mathcal{A}$, which form a directed acyclic graph (DAG) $G(\mathcal{A},\mathcal{E})$ going from the input to the output data atom. An atom may correspond to a filter in a convolutional layer, an attention head in a transformer, and/or predefined blocks of the DNNs layers. An edge, e.g., $(a_1,a_4){\in}\mathcal{E}$, corresponds to a connection between the outputs of a computational atom, say a_1, and the inputs to a subsequent computational atom, say a_4. Similarly, the input data atom a_{inp} in layer $\mathcal{L}_0$ may be connected to a computational atom in layer $\mathcal{L}_1$. A computational atom that has multiple incoming edges, e.g., a_7 with incoming edges (a_4,a_7) and (a_5,a_7), has as its input a weighted sum of the outputs of atoms a_4 and a_5.

Derived task-specific DNN architectures. Given a parent DNN architecture as described above, our aim is to determine, for each task t, a task-specific derived DNN architecture – an example is shown in Fig. 1 (right). The derived architecture is specified by a subset of computational atoms $\mathcal{A}^t$ where $\mathcal{A}^t{\subset}\mathcal{A}$ and the connectivity $\mathcal{E}^t$ is inherited from the parent DNN, i.e., $\mathcal{E}^t{=}\{\mathcal{A}^t{\times}\mathcal{A}^t\}{\cap}\mathcal{E}$. In other words, the set of computational atoms should induce a connected DAG from the input to the output data atoms, and correspond to computational atoms in the parent DNN architecture that will be used when performing inference task t. Note that such a derived task-specific architecture will have nonempty layers of computational atoms $\mathcal{L}_l^t{=}\mathcal{A}^t{\cap}\mathcal{L}_l$ for $l{=}1,\ldots,L{+}1$. For any computational or output atom a, we let $\mathcal{P}^t(a)$ denote the atoms that precede it in the previous layer.

As described in more detail in the sequel, the derivation of task-specific DNNs involves determining not only a collection of task-specific DAGs $\{G(\mathcal{A}^t,\mathcal{E}^t) : t{\in}\mathcal{T}\}$ from the parent DNN but also the connectivity in their union, possibly including adaptation layers [7]. Further atoms, which will not be part of the DAG associated with a given task, will not be used during inference for that task, thus resulting in a task-specific reduction in the computational costs.

Task loads and requirements. For each task $t{\in}\mathcal{T}$, let $\lambda(t)$ denote the rate at which task t inference requests arrive. Also, $q^{\min}(t)$ denotes the target inference quality (e.g., classification accuracy or MSE loss) for such tasks, and $\delta^{\max}(t)$ the maximum tolerable inference time associated with task t. Meeting these requirements will involve creating and training a shared DNN so that all derived task-specific DNNs fulfill the target values while possibly being executed across nodes with heterogeneous compute and network resources.

Orchestration: mapping task-specific DNNs to computational resources. We envisage that a network orchestrator will map input/output data atoms and computational atoms in a task-specific DNN to a set of nodes $\mathcal{N}$, corresponding to the physical compute resources in the mobile edge where (possibly) distributed inference can take place. We denote such a mapping to resources for task t via a function $d^t : \mathcal{A}^t{\cup}\{a_{\text{inp}},a_{\text{out}}\}{\to}\mathcal{N}$ where $d^t(a_{\text{inp}})$ denotes at which resource task t input data is first available. Also, $d^t(a_{\text{out}})$ denotes where the output atom of the task should be delivered, and $d^t(a)$ denotes at which node atom $a{\in}\mathcal{A}^t$ will be computed. Note that mappings associated with different tasks may map the same atom, i.e., an atom shared by different task-specific DNNs, to different compute nodes.

Given such a mapping $d^t()$ and further specification of compute and communication delays each task will experience, one can determine the overall execution time for each task: specifically, given the set of layers $\mathcal{L}^t{=}\{\mathcal{L}_l^t : l{=}0,\ldots,L\}$ and mapping $d^t()$ associated with task t, we determine the overall task execution time $\Delta(\mathcal{L}^t,d^t(\cdot))$.

Inference time constraints. The inference time constraints on the orchestrator's mappings are as follows:

$$\Delta(\mathcal{L}^t,d^t(\cdot)) \le \delta^{\max}(t) \quad \forall t \in \mathcal{T}. \tag{1}$$

Node memory constraints. The memory constraints on the orchestrator's mappings are given by:

$$\sum_{a\in\{a\,:\,\exists t\in\mathcal{T}\text{ s.t. }d^t(a)=n\}} m(a) \le \mu^{\max}(n) \quad \forall n \in \mathcal{N} \tag{2}$$

where $\mu^{\max}(n)$ denotes the maximum memory resources available at node n and $m(a)$ denotes the memory requirements of atom a.

Learning quality constraints. We further have constraints on the minimum inference quality achieved for the trained task-specific DNNs. We denote by $q(\mathcal{A}^t, t)$ the quality obtained when task t is executed on a task-specific DNN including the atoms $\mathcal{A}^t$. Note that the quality will also depend on how the training is conducted, as discussed in the sequel. For all $t \in \mathcal{T}$, we require:

$$q(\mathcal{A}^t, t) \geq q^{\min}(t), \quad \forall t \in \mathcal{T}. \tag{3}$$

Overall cost and optimization. We denote a collection of task-specific DNNs and associated resource mappings by:

$$\mathcal{M} = \{(\mathcal{A}^t, d^t()) : t \in \mathcal{T}\},$$

and the set of feasible $\mathcal{M}$'s by $\mathcal{F}$. Then the overall cost is:

$$c(\mathcal{M}) = \sum_{t \in \mathcal{T}} \lambda(t) \left\{ \sum_{a \in \mathcal{A}^t} [\kappa^{\text{comp}}(a, d^t(a)) + \right.$$
$$\left. + \sum_{a' \in \mathcal{P}^t(a)} \kappa^{\text{net}}(d^t(a'), a, d^t(a))] \right\} \tag{4}$$

where $\kappa^{\text{comp}}(a, n)$ is the cost due to executing atom a at node $d^t(a)$, and $\kappa^{\text{net}}(n', a, n)$ is the communication cost due to transferring the data required as input at atom a from $n'{=}d^t(a')$ to $n{=}d^t(a)$. Also, we recall that $\lambda(t)$ is the inference requests rate for task t, representing task popularity.

The optimization problem is then specified as follows:

$$\min_{\mathcal{M} \in \mathcal{F}} \{c(\mathcal{M}) : \text{Eqs. } (1)(2) \text{ and } (3)\}. \tag{5}$$

III. The TIDE Solution

Our TIDE framework provides high-quality solutions to the above problem with a low computational complexity. As shown in Fig. 2, it consists of five main steps:

1) Initial *training of the parent DNN*, where the architecture including *all* atoms is trained over *all* tasks;
2) Creation of efficient, *task-specific DNNs*, which in turn entails three sub-steps – derivation of task-specific DNN architectures, their mapping to physical nodes, further refinement of these decisions;
3) Additional, *task-specific training* of the task-specific DNNs obtained at the previous step;
4) *Deployment* of the task-specific DNNs on (possibly) different physical nodes;
5) Task-specific *inference execution*.

Steps 1 and 3–5 (purple boxes in Fig. 2) represent conventional steps in DNN training and inference. TIDE's key actions lie in Step 2, optimizing the creation of task-specific DNNs and mapping them to physical nodes. In making such decisions, TIDE exploits two key considerations that hold in the multi-task scenario we target. First, we consider that, if the parent DNN is adequately and sufficiently trained, then all tasks will meet their inference quality constraint (as in (3)) on the parent DNN. Second, and consistent with the consideration above, whenever constraint (3) is met for a candidate DNN for a given task, then if we add atoms to such a candidate, (3) will still hold – although doing so may increase the duration of the training process and/or the inference cost.

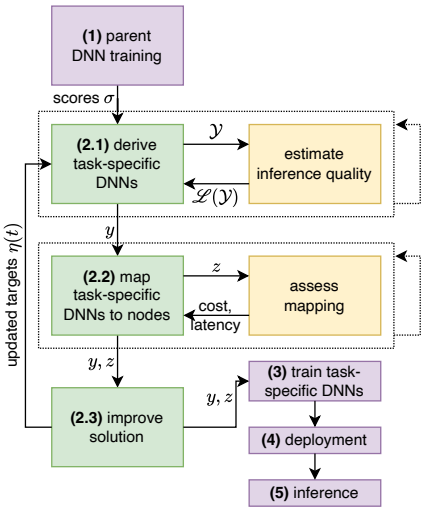

Fig. 2. TIDE schematics: conventional blocks for training/distributed inference (purple), TIDE's key steps (green), information used by TIDE (yellow).

Below we describe TIDE's steps towards determining the $\mathcal{M}$, i.e., the $\mathcal{A}^t$ and $d^t()$ decisions for each task t, that minimizes the inference cost based on an equivalent set of binary variables. For all $a \in \mathcal{A}, t \in \mathcal{T}$, and $n \in \mathcal{N}$, we let

- $y(a, t)$ denote whether task t uses atom a;
- $z(a, n, t)$ denote whether the instance of atom a used for the task t is mapped to node n.

We also denote by $\mathcal{Y} = \{(a, t) \in \mathcal{A} \times \mathcal{T} : y(a, t) = 1\}$ the set of task-to-atom assignments.

Step 1 – Parent DNN training: In this step, we train the full parent DNN in a joint manner, i.e., including all of the atoms in $\mathcal{A}$ for all tasks, using samples from all tasks. During this training, we evaluate a set of *scores* $\sigma(a, t)$, capturing how important each atom a is for each task t. For concreteness, as scores, we consider the relevance metric, computed through layer-wise relevance propagation [8]. The initial training continues until the scores are stable.

Step 2.1 – Deriving task-specific DNN architectures: The next step is the most critical and complex one in the TIDE framework. The high-level goal is to establish a task-to-atom assignment, i.e., setting the y-variables in our problem formulation. By doing so, we jointly account for two major aspects: *(i)* the inference quality $q(\mathcal{A}^t, t)$, i.e., whether we meet constraint (3) for all tasks, and *(ii)* the need to produce compact task-specific DNNs, which include as few atoms as possible. To this end, given the scores $\sigma(a, t)$ from Step 1, we derive the task-specific DNNs through a greedy, yet effective, approach. Before describing our algorithm, we introduce two functions that characterize a possible task-atom association $\mathcal{Y}$, namely, the value function $\mathsf{val}(\mathcal{Y})$ and the selector $\mathcal{L}(\mathcal{Y})$. The former is used to evaluate the benefit of a given $\mathcal{Y}$ in terms of both inference quality and task-specific DNN size. The latter, instead, states whether $\mathcal{Y}$ is guaranteed to meet the inference quality target.

For each task-to-atom assignment candidate decision $\mathcal{Y}$, the value $\mathsf{val}(\mathcal{Y})$ is defined as:

$$\mathsf{val}(\mathcal{Y}) = H(\mathcal{Y}) - \sum_{t \in \mathcal{T}} [A_{\mathcal{Y}}(t) - \eta(t)]^{+} \tag{6}$$

where the *entropy* $H(\mathcal{Y})$ of an assignment $\mathcal{Y}$ is given below, $A_{\mathcal{Y}}(t)$ denotes the number of atoms in the assignment, $\eta(t)$ denotes the target number of atoms to assign to task t, and $[z]^{+}=\max(0,z)$. The target number $\eta(t)$ is initialized to $\eta(t)=|\mathcal{A}|$ for all tasks, and then adjusted in Step 2.3 to further compress the task-specific DNNs, as detailed later.

Definition of the entropy $H(\mathcal{Y})$ of the assignment $\mathcal{Y}$. We define the entropy $H(\mathcal{Y})$, associated with a task-to-atom assignment $\mathcal{Y}\subseteq\mathcal{A}\times\mathcal{T}$, using an approach inspired by [9] and based on three main steps: *(i)* we create an undirected weighted graph based on the task-to-atom assignment $\mathcal{Y}$, *(ii)* we consider a weighted random walk on the graph, and *(iii)* we compute the entropy rate $H(\mathcal{Y})$ of the random walk as per [9]. Notice how a higher entropy rate corresponds to an increasing number of atoms used by a task, hence, to a higher inference quality. However, this may make the task-specific DNNs too large. Accordingly, the second term in (6) acts as a lever to prevent *too many* atoms being associated with a certain task: tasks getting more than $\eta(t)$ atoms incur a penalty.

Regarding the selector function $\mathscr{L}(\mathcal{Y})\rightarrow\{0,1\}$, this is used to assess whether $\mathcal{Y}$ meets the target inference quality, i.e., whether setting $y(a,t)=1$ for the atoms and tasks $(a,t)\in\mathcal{Y}\subseteq\mathcal{A}\times\mathcal{T}$ honors the constraint in (3) (top yellow block in Fig. 2).

Given the selector $\mathscr{L}(\mathcal{Y})$ and the value function $\mathsf{val}(\mathcal{Y})$, as well as the targets $\eta(t)$, TIDE follows an iterative procedure: it starts from a situation where all tasks use all atoms, and disables, at each iteration, the task-to-atom association that degrades the value of $\mathcal{Y}$ (as defined in Eq. (6)) the least while preserving the target inference quality.

Step 2.2 – Planning and assessing the mapping of task-specific DNNs to nodes: The goal of this step is to map atoms to nodes, i.e., setting the $z(a,n,t)$ variables. We remark that this operation entails no ML-related decisions; it follows that atom-to-node mappings can be assimilated to a pure virtual network function (VNF)-chain placement problem, where: *(i)* VNFs to place correspond to atoms; *(ii)* the virtual machines where they can be placed correspond to nodes; *(iii)* the traffic sent from a VNF to another corresponds to the input size of the (receiving) atom. Thanks to the this mapping, represented in the middle yellow block in Fig. 2, we can exploit any existing VNF placement algorithm [10] and integrate it within TIDE.

Step 2.3 – Refining the decisions: TIDE now refines decisions by trying to make task-specific DNNs as small as possible – in other words, to set some $y(a,t)$ and $z(a,n,t)$ variables to zero. To do so, it first quantifies the inference cost $c(\mathcal{M})$ associated with each task t by considering the mapping of the atoms it uses, the sample arrival rate, and the computing and communication costs (as in Eq. (4)). It then starts from the task $t^{\star}$ associated with the highest inference cost, reduces its target number of atoms $\eta(t^{\star})$ by one, and re-runs the task-specific DNN derivation (Step 2.1 of TIDE). If, at the end of Step 2.1, the number of atoms associated with $t^{\star}$ is still higher than $\eta(t^{\star})$, it means that it is impossible to further reduce the number of atoms assigned to that task; thus, it blacklists $t^{\star}$ and does not consider it again. Once all tasks are blacklisted, no further refinements are possible, and Step 2.3 is completed.

Step 3 – Per-task DNN training: The obtained task-specific DNN architectures are now trained until the required inference quality, $q^{\min}(t)$ is achieved. As mentioned, during such training, samples related to tasks that share atoms in their task-specific DNNs may "pull" the weights in different directions. Atoms that have not been selected for a given task-specific DNN will be frozen.

Step 4 – Task-specific DNNs deployment: The trained atoms are then deployed on (possibly, different) physical nodes, according to the decisions $z(a,n,t)$ made in Step 2.2. TIDE thus applies computational split and implements distributed inference, as needed for minimizing inference costs.

Step 5 – Inference: Each trained and deployed task-specific DNN can now perform inference, processing the task samples as they arrive according to the rate $\lambda(t)$. The overall cost $c(\mathcal{M})$ incurred by the inference tasks is given in Eq. (4).

IV. Experimental Results

A. Reference scenario and benchmarks

Dataset and DNN. For our performance evaluation, we consider a computer vision scenario, where the AlexNet DNN is used to perform image classification over the CIFAR-100 dataset. CIFAR-100 includes 100 fine-grained classes, grouped into 20 coarse-grained classes; we leverage this structure by creating 10 tasks, each including two coarse-grained classes (hence, 10 fine-grained ones), i.e., 10% of the whole dataset. To each task we assign a random popularity level, extracted from a Zipf distribution with shape parameter $\alpha=2$ [11].

Atoms and nodes. We create a computational atom for each of the 448 filters of AlexNet's convolutional layers, and use the `calflops` library [12] to estimate their computational complexity (in MFLOPs). We then use the complexity of the whole AlexNet as our cost normalization unit.

Targets and scores. We take as a reference a centralized inference time of 50 ms per sample [13], and set the task delay limit to 200 ms. As for the quality target, we fix it to a common value for all tasks (i.e., $q^{\min}(t)=q^{\min}$). As AlexNet's best-case accuracy [14] is approx. 70% over the CIFAR-100 dataset, we then let the accuracy target, $q^{\min}$, vary between 0.5 and 0.7. To compute the $\sigma(a,t)$ scores, we use the layer-wise relevance metric [8] and stop the parent DNN training after all relevance values stabilize within $\epsilon=10\%$ of their final values (after 8 epochs in our case). Finally, we implement the selector function $\mathscr{L}()$ using the library in [15], which implements the accuracy estimation methodology [16].

Benchmarks. We compare TIDE to the following schemes:

• *"prune"*: after training the parent DNN, the original AlexNet is pruned in a structured manner, keeping the highest-magnitude filters and setting a pruning fraction between 50% and 95% depending on the accuracy target. The pruned DNN

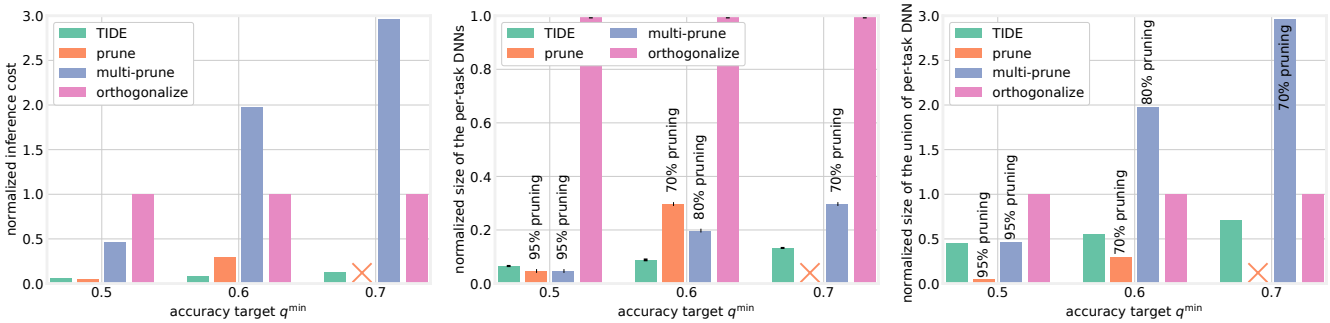

Fig. 3. TIDE vs. its benchmarks: total inference cost (left), average (bars) and minimum and maximum (black lines) size of the per-task DNNs (center) and of their union (right), for different levels of the quality target $q^{\min}$.

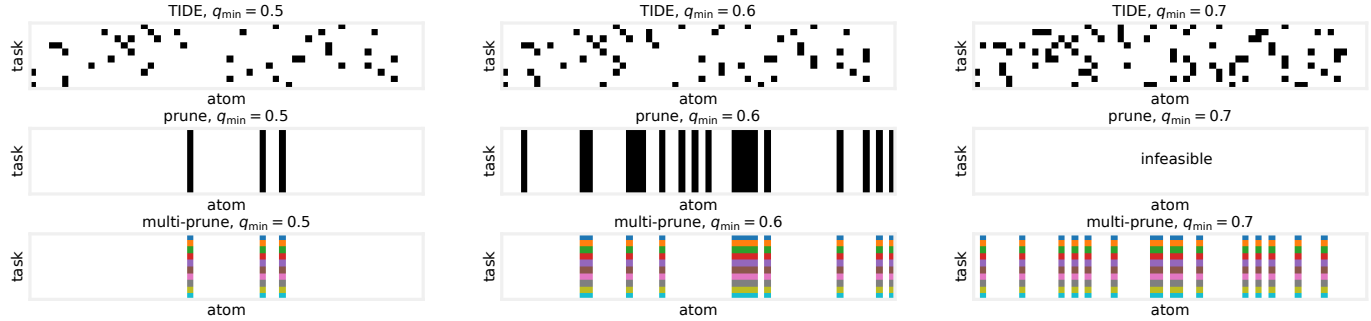

Fig. 4. Visual representation of the atom-to-task associations used by task-specific DNNs, for different strategies (rows) and values of the quality target $q^{\min}$ (columns). Each atom-task cell is colored if used; different colors correspond to different instances deployed for the same atom. The plots show how TIDE's task specific DNNs use a lower number of atoms as compared to its alternatives and that atoms shared by multiple tasks end up being deployed only once on physical resources, while multi-prune crates as multiple instances of such common atoms as the number of tasks sharing them.

is then fine-tuned and used for all tasks. Upon deployment, split computing can be exploited;

• *"multi-prune"*: proceeds in the same manner, but creates a separate DNN (with the same pruning fraction) for each task;

• *"orthogonalize"*: it uses the parent DNN for all tasks, but applies the methodology in [17] to orthogonalize gradients so as to minimize interference among tasks.

We note that, for fairness of comparison, for the "prune" and "multi-prune" strategies we tried a range of pruning fractions, and then apply the one resulting in the cheapest solution.

B. Numerical results

The first aspect we are interested in is whether TIDE is better than the benchmarks at minimizing inference cost, i.e., the objective in (5). Fig. 3(left) depicts the total inference cost obtained by different strategies, normalized to the cost incurred by the centralized approach where the full AlexNet is executed at one node. Such a node has the memory and computing capacity necessary to run the full AlexNet. As can be seen, the solutions obtained by TIDE result in a very low cost, indeed much lower than its counterparts. We can also observe that, as $q^{\min}$ increases, the costs of the "prune" and "multi-prune" strategies grow much faster than TIDE's. Also, notice how "prune" is unable to achieve the highest quality level (namely, $q^{\min}$=0.7), as indicated by the orange cross. Finally, "orthogonalize" cannot adapt the deployed DNN to the quality target, hence, its cost is the same as for the centralized solution (i.e., the normalized cost is equal to 1).

Fig. 3(center) depicts the size of the per-task DNNs (average as well as minimum and maximum values), quantified through the number of parameters and normalized to that of the full AlexNet. It is possible to observe, consistently with Fig. 3(left), that TIDE's DNNs are always smaller than those created by the benchmarks. The figure also shows the pruning fraction selected under the "prune" and "multi-prune" strategies: higher values of $q^{\min}$ are associated with larger DNNs, hence, smaller pruning fractions. Also, "multi-prune" can use a higher pruning fraction than "prune" for $q^{\min}$=0.6, since "multi-prune" task-specific DNNs are fine-tuned separately. Fig. 3(right) depicts the size of the *union* of all task-specific DNNs. We note that TIDE might yield more parameters than "prune": indeed, the DNN created by "prune" is shared, in its entirety, by all tasks, while TIDE's task-specific DNNs may overlap only partially. Finally, we notice how TIDE yields task-specific DNNs of very similar size across all tasks.

We now drop the "orthogonalize" benchmark and present in Fig. 4 a visual representation of the atom-to-task associations chosen by the different strategies. In each plot, rows correspond to tasks, columns to atoms, and black/colored "cells" to active associations. It can be seen that TIDE (top row of the sub-plots) follows no fixed pattern in choosing the associations. "Prune" (middle row), instead, associates the atoms it chooses with all tasks. "Multi-prune" (bottom row) activates the same atoms for all tasks, but associates with each task a different instance of the same atom – hence, we denote the atoms using different colors. Moreover, TIDE can adapt the number of atoms used by each task to the task itself, unlike "prune" and "multi-prune".

V. Related work

A first area TIDE is related to concerns DNNs targeting multiple tasks. In this context, approaches such as those in [3], [4] envision combining shared and task-specific parts (e.g., layers, stems, branches) within the same DNN architecture. Later approaches [18], [19] target dynamic scenarios, where tasks appear sequentially, and envision dynamically expanding the DNN architecture. Compared to these approaches, TIDE aims at better balancing the goal of adapting to multiple tasks with the need to keep the DNN size manageable.

An alternative approach to multi-task DNN training and inference is in [17], which seeks to prevent catastrophic forgetting in sequential scenarios by making the gradient steps of the current task orthogonal to those of previous ones. However, [17] does not aim at making a DNN architecture suitable for distributed inference, thus it misses the optimization opportunities that TIDE exploits. Studies that instead aim at DNNs distributed deployments try to balance the training and inference performance against the available hardware and the associated costs. Examples include generic approaches like DNN pruning [20], [21] and quantization [22], as well as works envisioning a mutual adaptation between DNN architecture and existing resources [23]. In contrast to these works, TIDE makes more fine-grained decisions about (i) the elements of the DNN architecture that should be used, (ii) by which tasks, and (iii) where to deploy them.

Finally, TIDE is related to the area of Network Architecture Search (NAS) and, more specifically, to techniques to predict the DNNs training or inference performance. Many approaches find convergence bounds [24]; others are learning-curve based (i.e., they examine the first few training epochs and predict the loss and accuracy evolution) [25], or model-based, requiring no *a-priori* information [26]. Some recent works [16] even have open-source implementations [15].

VI. Conclusions

We considered the execution at the mobile edge of inference tasks that are homogeneous in nature, but heterogeneous in their input sample distribution, which may also change over time. We introduced a novel approach based on the partitioning of a parent DNN into task-specific networks, each composed of interconnected computational elements of the parent DNN. To create task-specific DNNs, we proposed TIDE, an algorithmic framework that leverages metrics computed during training, an effective matching algorithm, and an optimized deployment of task-specific DNNs at the edge. Our results show that TIDE reduces inference cost and time by 90% and 80% (resp.) when compared to centralized approaches, and by over 60% and 70% (resp.) relatively to the best benchmark.

Acknowledgement

This work was partially funded by the SNS JU under the EU's Horizon Europe research and innovation programme under the MultiX project Grant Agreement No. 101192521 and in part by NSF Award CNS- 2212202.

References

[1] F. Malandrino, C. F. Chiasserini, and G. Di Giacomo, "Efficient distributed dnns in the mobile-edge-cloud continuum," *IEEE/ACM Trans. on Networking*, 2022.

[2] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Trans. on Parallel and Distributed Systems*, 2021.

[3] X. Liu *et al.*, "Multi-task deep neural networks for natural language understanding," in *ACL*, 2019.

[4] A. V. Malawade, T. Mortlock, and M. A. Al Faruque, "HydraFusion: Context-aware selective sensor fusion for robust and efficient autonomous vehicle perception," in *ACM/IEEE ICCPS*, 2022.

[5] D. Khashabi *et al.*, "UNIFIEDQA: Crossing format boundaries with a single QA system," in *ACL*, T. Cohn, Y. He, and Y. Liu, Eds., 2020.

[6] P. Rajpurkar and et al., "Deep learning for chest radiograph diagnosis: A retrospective comparison of the CheXNeXt algorithm to practicing radiologists," *PloS Med.*, vol. 15, no. 11, 2018.

[7] T. K. Johnsen, I. Harshbarger, and M. Levorato, "An overview of adaptive dynamic deep neural networks via slimmable and gated architectures," in *IEEE ICTC*. IEEE, 2024.

[8] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PloS one*, 2015.

[9] M.-Y. Liu, O. Tuzel, S. Ramalingam, and R. Chellappa, "Entropy-rate clustering: Cluster analysis via maximizing a submodular function subject to a matroid constraint," *IEEE TPAMI*, 2013.

[10] G. Sallam *et al.*, "Joint placement and allocation of vnf nodes with budget and capacity constraints," *IEEE/ACM Trans. on Networking*, 2021.

[11] Y. Wang, Y. Tong, D. Shi, and K. Xu, "An efficient approach for cross-silo federated learning to rank," in *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1128–1139.

[12] X. Ye. (2023) calflops: a FLOPs and Params calculate tool for neural networks in pytorch framework. [Online]. Available: https://github.com/MrYxJ/calculate-flops.pytorch

[13] A. M. Wahid, T. Hariguna, and G. Karyono, "Optimizing feature extraction for website visuals: A comparative study of alexnet and inception v3," in *IEEE CITSM*, 2024.

[14] N. Sharma, V. Jain, and A. Mishra, "An analysis of convolutional neural networks for image classification," *Procedia computer science*, 2018.

[15] M. Ruchte, A. Zela, J. Siems, J. Grabocka, and F. Hutter, "Naslib: A modular and flexible neural architecture search library," https://github.com/automl/NASLib, 2020.

[16] Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter, "Nas-bench-suite: Nas evaluation is (now) surprisingly easy," in *ICLR*, 2022.

[17] G. Saha, I. Garg, and K. Roy, "Gradient projection memory for continual learning," in *ICLR*, 2021.

[18] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," in *ICLR*, 2018.

[19] J. Yoon *et al.*, "Scalable and order-robust continual learn- ing with additive parameter decomposition," in *ICLR*, 2020.

[20] W. Kwon, S. Kim, M. W. Mahoney, J. Hassoun, K. Keutzer, and A. Gholami, "A fast post-training pruning framework for transformers," *NeurIPS*, 2022.

[21] K. Azarian, Y. S. Bhalgat, J. Lee, and T. Blankevoort, "Learned threshold pruning," in *ICLR*, 2021.

[22] B. Rokh, A. Azarpeyvand, and A. Khanteymoori, "A comprehensive survey on model quantization for deep neural networks in image classification," *ACM Trans. on Intelligent Systems and Technology*, 2023.

[23] C. Singhal, Y. Wu, F. Malandrino, M. Levorato, and C. F. Chiasserini, "Resource-aware deployment of dynamic dnns over multi-tiered interconnected systems," in *IEEE INFOCOM*, 2024.

[24] X. Li, Z. Song, R. Tao, and G. Zhang, "A convergence theory for federated average: Beyond smoothness," in *IEEE Big Data*, 2022.

[25] C. Jiang, Z. Huang, T. Pedapati, P.-Y. Chen, Y. Sun, and J. Gao, "Network properties determine neural network performance," *Nature Communications*, 2024.

[26] X. Ning, Y. Zheng, T. Zhao, Y. Wang, and H. Yang, "A generic graph-based neural architecture encoding scheme for predictor-based NAS," in *European Conference on Computer Vision*, 2020.