Regularized Bottleneck with Early Labeling

Gabriele Castellano, Fabio Pianese, Damiano Carra, Tianzhu Zhang, Giovanni Neglia

Regularized Bottleneck with Early Labeling

Gabriele Castellano*†, Fabio Pianese*, Damiano Carra‡, Tianzhu Zhang*, Giovanni Neglia†
* Nokia Bell Labs, France, {name.surname}@nokia-bell-labs.com,
† Inria, Université Côte d'Azur, France, {name.surname}@inria.fr,
‡ Università di Verona, Italy, {name.surname}@univr.it

***Abstract*—Small IoT devices, such as drones and lightweight battery-powered robots, are emerging as a major platform for the deployment of AI/ML capabilities. Autonomous and semi-autonomous device operation relies on the systematic use of deep neural network models for solving complex tasks, such as image classification. The challenging restrictions of these devices in terms of computing capabilities, network connectivity, and power consumption are the main limits to the accuracy of latency-sensitive inferences. This paper presents ReBEL, a split computing architecture enabling the dynamic remote offload of partial computations or, in alternative, a local approximate labeling based on a jointly-trained classifier. Our approach combines elements of head network distillation, early exit classification, and bottleneck injection with the goal of reducing the average end-to-end latency of AI/ML inference on constrained IoT devices.**

I. Introduction

In recent years, an increasing number of tasks (from image classification, to image segmentation, to recommendation) is performed using Deep Neural Networks (DNNs). The success of DNN models reflects their increasing accuracy in a large variety of scenarios. Nevertheless, their complexity and huge over-parameterization constitute a significant burden from the computational viewpoint. In several IoT scenarios, the data to process are collected in a resource-constrained device (e.g., a picture taken from a drone) in which a complex DNN model could not fit. Data need to be transferred over the network to a remote, powerful server that can run large DNN models. In this scenario, the overall inference latency depends, in addition to the server processing time, on the data transfer time. In case of degraded transmission channel conditions, this may lead to an excessive end-to-end delay.

Split Computing [1] proposes to divide large DNN models into two portions, deployed at the end device and at the remote server respectively. In this way, part of the computational load is carried locally, and the activation values at the split point are transmitted to the remote server. In addition to the obfuscation of the intermediate representation, which is not immediately recognizable by human observers as the original input would be, this approach unlocks compelling trade-offs between the delay and the amount of data to transmit, provided that the split point is chosen carefully. In fact, for many off-the-shelf DNN architectures such as ResNet [2] and VGG, the size of the representations at the first layers may exceed the size of the input data. To reduce communication overhead, the network would need to be split at a later stage, i.e., closer to the output, thus leaving most of the computation to the mobile device [3].

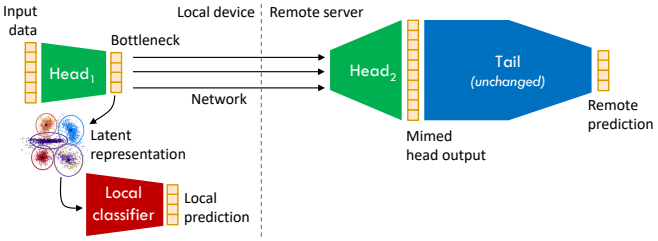


Fig. 1: The ReBEL architecture. Local IoT device runs Head_1 and a classifier. If the confidence of the early classification is not sufficiently high, the data at the bottleneck is sent to the remote server that runs Head_2 and Tail.

In order to take advantage of the Split Computing approach, the solutions aiming at decreasing the delay fall into two categories: *early-exit* and *bottleneck injection*. With early-exit [4], the DNN is augmented with parallel branches which operate on the activation values of intermediate layers to provide tentative inference results. If the confidence of the early inference is sufficiently high, the data is not transmitted to the server, with a reduction of the average inference time. With bottleneck injection [1], the DNN is modified to explicitly include a small-size layer at the beginning, whose output is sent to the server. This has an effect on the accuracy, since such a smaller intermediate representation may be less rich than the original DNN.

In this paper we propose *Regularized Bottleneck with Early Labeling* (ReBEL), an improved split computing architecture that combines elements of early-exit and bottleneck injection for dynamically offloading image classification tasks in IoT deployments (Fig. 1). ReBEL needs to address multiple challenges in order to increase the *expressivity* of the bottleneck interface: its representation should be compact enough for efficient transmission to the remote server, informative enough to enable high accuracy inference at the server, yet sufficiently simple to be correctly processed by the early classifier (Sec. III). We outline a method for training ResNet-family split-execution DNNs which accommodates these somewhat conflicting goals and is general enough to be extended to other DNN architectures (Sec. IV). We introduce the use of Gaussian mixtures as early classifiers (GMC), providing a boost in the accuracy achievable after minimal local processing (Sec. V). Our experiments (Sec. VI) show that the GMC achieves a significant gain in accuracy in comparison to commonly used linear layers, $K-$*Nearest Neighbors*, or $K-$*means*. In particular, experiments on the CIFAR-100 dataset indicate that ReBEL provides considerable benefits compared to existing

approaches and can enable a set of practically interesting trade-offs between overall inference accuracy and end-to-end latency with a minimal computational burden on the IoT device.

II. Related Work

This section explores existing proposals for executing state-of-the-art AI models leveraging the network while achieving acceptable end-to-end latency and inference accuracy on resource-constrained devices.

Split Execution. Split execution aims at slicing large DNN models into multiple sections and offloading parts of the computation to powerful remote servers [5]. The base DNN model can be split by layers, channels, or by parallelizing input data. Splitting does not incur into accuracy losses since it merely distributes the processing workload. Neurosurgeon [6] is the first framework exploiting model splitting, partitioning models layer-wise based on the cost of computation, and enabling the scheduling of split models among mobile devices and infrastructure. Similarly, Li et al. [7] partitions models into layers using an auto-tuning mechanism based on performance. JointDNN [8] supports collaborative model training and inference between the mobile device and the servers, describing the operation of DNN layers as an integer linear program (ILP) modeling battery constraints, processing load, and quality of service parameters. JALAD [9] also formulates model splitting as an ILP problem aiming to minimize the end-to-end latency. IONN [10] partitions a DNN model and employs incremental offloading to realize energy-efficient collaborative inference between mobile devices and the edge server. Scission [11] and Couper [12] systematically adopt layer-wise DNN partitioning and workload scheduling. DeepThings [13] employs a fuse-tile partitioning algorithm to reduce the incurred memory footprint. Zhou et al. [14] propose a containerized spatial partitioning runtime to accelerate CNN inference for IoT applications at the edge.

In all the above works, the DNN architecture is not modified. Therefore, the transmission of the intermediate output may require a very high amount of data — the choice of the splitting point is limited by the architecture. In our proposal, instead, we modify the DNN architecture to include a bottleneck that limits the amount of data to be transmitted.

Early-exit. Early-exit is an adaptive inference method that introduces one or more intermediate branch classifiers to a DNN model in the training process. An early-exit decision can thus short-cut the inference process, resulting in a faster outcome and a reduced workload (no need to process the remaining DNN layers) — such an approach was originally proposed for large models running on servers, with no connection with the split computing approach. The idea was first proposed in [15] as "cascading neural networks", where intermediate output layers are added to an off-the-shelf DNN and trained while keeping the weights of the DNN frozen. In the past years, the early-exit principle has been proposed under different names: conditional deep learning [16], shallow-deep networks [17], adaptive inference [18], budgeted prediction technique [19], etc. BranchyNet [4] achieves 2-6x inference speedups by jointly optimizing the branch structures, exit criteria, and the loss functions for all the exit points. Neshatpour et al. [20] arrange Convolutional Neural Networks (CNNs) into multiple stages with contextual awareness of the input workload to achieve a desirable energy-accuracy trade-off. Dynexit [21] employs a dynamic strategy to update the loss weight of each exit point for ResNets [2].

DDNNs [22] and DeepIns [23] extend early-exit to a distributed computing hierarchy (cloud, edge, and IoT devices). Similarly, Dynamic offload [3] improves on split execution by allowing some of the inferences to be decided in an early-exit fashion on the local IoT device, without systematically carrying forward the entire computation on the remote platform. Lo et al. [3] split a DNN model into an on-device auxiliary DNN and a remote principal DNN, allowing dynamic offload of the workload based on an early assessment of the *confidence* in the early prediction's accuracy, which is based on a softmax classification layer. Unlike [4] and [16], confidence thresholds are defined for each class pair (or per class, when the number of classes is large) at training time and determine the tensor arrays that need to be sent over the network to the principal DNN. Edgent [24], Boomerang [25], and DDI [26] combine early-exit and model splitting to achieve the best trade-offs between latency and accuracy, using a regression model to predict the cross-layer inference latency due to offloading.

In the above works, the intermediate data that may be sent (depending on the confidence of the early inference) may still be very large, while in our case we explicitly inject a bottleneck to avoid transmitting a large amount of data.

Bottleneck Injection. Limiting the network footprint of distributed DNN operation is a critical aspect in an IoT system, as it can both reduce transmission latency and increase device workload. One option is to use a set of additional layers to compress (code and decode) the representation of a given layer — the compression could be lossy [27], channel-agnostic [28], or channel-aware [29]. Such an approach, nevertheless, increases the computational burden for coding and decoding, without advancing the inference computation. Another option is *bottleneck injection*, which consists in building new DNN models that contain a thin intermediate layer at the offloading point. This enables the seamless integration of compression *within* a distributed DNN inference at the cost of a decreased accuracy. Head Network Distillation (HND) [30] introduces bottleneck injection based on the observation that pre-trained DNN models often have redundant parameters in the first layers. HND uses knowledge distillation [31] to build a compact head network replacement (student) mimicking the original model's head (teacher) behaviors. BottleFit [32] improves on the HND approach for training small and accurate split models to be deployed on constrained IoT devices at the network edge.

Differently from these works, we augment the bottleneck with an early-exit branch to provide early classification results in the case of simple inputs.

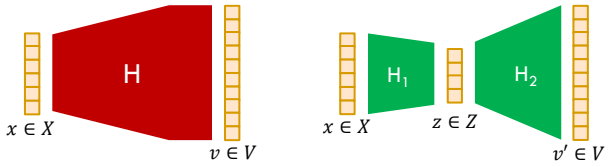


Fig. 2: The bottlenecked head (right) mimics the original head output (left).

III. ReBEL Distributed Architecture

In the design of ReBEL, we start from a pre-trained DNN model (e.g., ResNet, DenseNet), and we aim at deriving a modular model that is better adapted for a distributed deployment involving constrained devices and a network. Our goal is to overcome the intrinsic limitations of widespread computer vision models, where the intermediate output of early hidden layers is much bigger than the original data, preventing an effective split deployment. There are two key elements that characterize our approach: *(i)* we modify the early stages of the model (Head) by injecting a bottleneck as a natural split point between IoT device and remote server, and *(ii)* we exploit the relatively small bottleneck output by means of a local classifier that can predict labels early with a reliable confidence metric.

The components of the ReBEL architecture are shown in Fig. 1. Inspired by HND [33], we replace the first segment of the DNN model (the *Head*) with a simpler bipartite structure, namely Head_1 and Head_2, which delimit a bottleneck channel that carries the intermediate data (see Fig. 2). The *Tail* section of the model is left unchanged; the sequence of Head_1 and Head_2 is trained to mimic the behavior of the original Head so that the Tail keeps operating normally (in Fig. 2, v and v' should be as similar as possible). We process the Head_1 output, called z, in two steps. First, we use it on the local device via an early classifier. In case of low classification confidence, we transmit z to a more powerful machine such as an Edge server, where the remainder of the deep model computation (i.e., Head_2 and Tail) can take place.

A. Variational Head

In earlier architectures based on bottleneck injection [33], the output z of the bottleneck layer is used as it is. Instead, we consider a different approach, which we call *Variational Head*, inspired by Variational Autoencoders [34] — although the architecture we employ (Fig. 3) is based on convolutional layers that do not behave as ordinary autoencoders. In the following, we explicitly refer to the ResNet50 architecture; however, this methodology is general enough to be applied to any computer vision model.

We design a head network that serves images of size $3\times32\times32$ (i.e., the size of images in the CIFAR-100 dataset). Our variational head replaces the first seven residual blocks of ResNet50, i.e., a total of 24 convolutional layers, and is composed of a Head_1 and a Head_2, each with 4 convolutional layers (Fig. 3). We normalize the output of each convolutional layer by mean of Generalized Divisive Normalization (GDN) [35], a parametric nonlinear transformation that was shown to be well suited for "Gaussianizing" data from natural

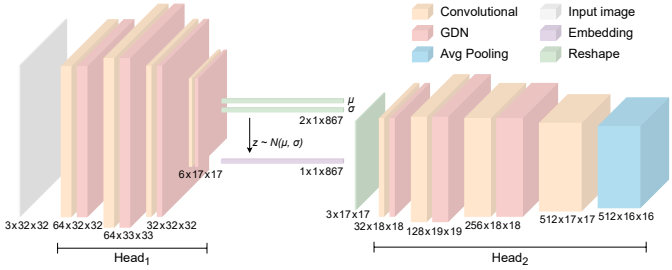


Fig. 3: Architecture of ReBEL variational head. Head_1 outputs two vectors μ and σ that are used to sample an embedding $z \sim N(\mu(x), \sigma(x))$. Head_2 further extracts features from z and produces an input for the Tail.

images. Significantly, Head_1 does not directly convert an image $x \in X$ into an embedding vector $z \in Z$. Instead it maps x into a distribution in the space Z. In practice, when processing x, Head_1 outputs two vectors: a mean vector $\mu(x) \in Z$ and a covariance vector $\sigma(x) \in Z$. Such vectors model a multivariate normal distribution $N(\mu(x), \sigma(x))$.[1] This distribution is called *posterior distribution* $p(z|x)$, and it is used to sample, given the input vector x, an embedding $z \sim N(\mu(x), \sigma(x))$ for the input vector x, which can then be sent to the local classifier and/or to Head_2. Although embeddings in the latent space Z are obtained through a shallow feature extraction process (Head_1 contains only four convolutional layers), we show that they can still be used to provide approximate predictions for a significant portion of the dataset. In Sec. IV, we show how this architecture enables a particular regularization process that leads to producing high-quality embeddings as output of Head_1.

Note that Head_2 does not feature deconvolutional layers: differently from an autoencoder, our goal is not to reconstruct the input after compression but to proceed in the feature extraction process. Therefore, the bottleneck output is further processed by a convolutional block and then sent to the Tail.

B. Inference Workflow

In summary, the inference workflow proceeds as follows. On the device, a sample $x \in X$ is forwarded to Head_1, which outputs vectors $\mu(x)$ and $\sigma(x)$. These are used to sample an embedding vector $z \sim N(\mu(x), \sigma(x))$ in the latent space Z. Such embedding is then fed to the local classifier, which produces *(i)* a local prediction, i.e., assigns a label $c' \in \{1, \ldots, C\}$ to the embedding z (and therefore to sample x), and *(ii)* a confidence value $\gamma \in [0, 1]$, that is a metric of how confident the local classifier is to assign label c' to sample x. If the confidence γ is higher than a predefined threshold, the prediction is considered reliable enough and the inference process terminates. Otherwise, the embedding z is sent to the remote server and fed to Head_2 and Tail, thus obtaining a new and more accurate label $c'' \in \{1, \ldots, C\}$.

IV. Model Training

ReBEL model training jointly optimizes 3 loss functions:

[1]Note that the covariance is supposed to be a square matrix. To reduce the number of parameters, we limit our variational output to distributions with diagonal covariance matrices, where vector $\sigma(x) \in Z$ represents the diagonal.

1. a *Mimic Loss*, which steers the Head to approximate the behavior of the original model (as featured in HND);
2. a *Regularization Loss*, which incentives Head$_1$ to generate good quality embeddings for classification;
3. an *Early Classification Loss*, which helps enhance the early classifier's success rate on the latent space.

In this section we first detail each of the loss functions, providing both the formulation and an intuition underpinning its design. Then we describe the process to optimize such functions jointly and train our distributed model.

A. Mimic Loss

The original DNN model can be seen as the composition of two functions: $H\colon X \to V$, i.e., the Head that maps input $x \in X$ to an intermediate vector $v \in V$ (see Fig. 2, left part), and $T\colon V \to Y$, i.e., the Tail model that maps the intermediate vector v to the prediction $c \in \{1, \ldots, C\}$. By applying HND, we replace H with $H_1 \circ H_2\colon X \to V$, i.e., the composition of the two functions $H_1\colon X \to Z$ and $H_2\colon Z \to V$. Fig. 2 shows that the sequence of the two models maps input x to a different intermediate vector $v' \in V$. The objective of ReBEL is to train H_1 and H_2 so that the distance between v and v' is minimized for every input x. This can be achieved by minimizing the Sum of Squared Error between outputs of the two models:

$$\mathcal{L}_M = \textstyle\sum_{x \in X} \|H_2(H_1(x)) - H(x)\|^2. \tag{1}$$

Note that this is similar to the reconstruction loss in autoencoders. However, rather than reconstructing the original input x, we herein try to produce an output similar to v, thus 'mimicking' the behavior of the original head.

B. Regularization Loss

Besides reproducing the results of the original Head, we need to render the bottleneck between H_1 and H_2 suitable for early classification, that is, making it so that points with the same label are found nearby in the latent space Z. As shown in [36], this is difficult to achieve at the early layers of a model (as shown in Fig. 3, our head features only 4 convolutional layers before the bottleneck). The classification task gets progressively easier the more high-level features have been extracted from the input image, which is indeed the reason why deep models manage to achieve impressive accuracy. Conscious of this challenge, in ReBEL we seek a regular structure for the latent space that successfully represents at least a portion of the data. The intuition behind this is that the full DNN model depth is only beneficial for the hardest samples in the datasets, while in many cases a shorter feature extraction process would be enough (see Fig. 4).

Our regularization approach takes inspiration from variational inference, a technique that approximates complex distributions by choosing a parameterized family of simpler ones and selecting the one that minimizes a given error measurement. In ReBEL, we approximate the posterior distribution $p(z|x)$ (i.e., the output of Head$_1$) with a prior $p(z)$ that is a mixture of Gaussians:

Fig. 4: Two images of a horse. The left one is easy to label even with a small model, whereas the right one requires a deeper network architecture [37].

$$p(z) = \textstyle\sum_{c=1}^{C} \sum_{k=1}^{K} \pi_{ck} \mathcal{N}(z; \mu_{ck}, \Sigma_{ck}), \tag{2}$$

where C is the number of labels (e.g., 100 in CIFAR-100 dataset) and K is the number of Gaussian components we assume for each label. π_{ck}, μ_{ck}, and Σ_{ck} are trainable parameters. In practice, these parameters are sampled from those of the early classifier (see Sec. VI).

We express the approximation error between the two distributions $p(z|x)$ and $p(z)$ with the Kullback–Leibler divergence [38]. Note that, by construction of the variational head, also $p(z|x)$ is a Gaussian distribution, mean $\mu(x)$ and covariance $\sigma(x)$ that are functions of the input x. The KL–divergence between two Gaussian distributions can be directly expressed in terms of their means and covariance matrices. Hence we define the regularization loss as

$$\begin{aligned}
\mathcal{L}_R &= KL(p(z|x), p(z)) \\
&= KL(\mathcal{N}(\mu(x), \sigma(x)), \textstyle\sum_{c,k} \pi_{ck} \mathcal{N}(z; \mu_{ck}, \Sigma_{ck})) \\
&= \textstyle\sum_{x \in X} \sum_{c,k} \frac{1}{2} \min_k \Big[\log \frac{|\Sigma_{ck}|}{|\sigma(x)|} - \dim(Z) \\
&\qquad + (\mu(x) - \mu_{ck})^T \Sigma_{ck}^{-1} (\mu(x) - \mu_{ck}) \\
&\qquad + \mathrm{tr}\{\Sigma_{ck}^{-1} \sigma(x)\} \Big] \cdot \mathbb{1}_{\{c = l(x)\}},
\end{aligned} \tag{3}$$

where $\mathrm{tr}\{\cdot\}$ is the trace operator, $l(x)$ is the correct label for sample x and, for brevity, we use vector $\sigma(x)$ to denote the diagonal covariance matrix $\mathrm{diag}(\sigma(x))$.

C. Early Classification Loss

The ReBEL local classifier should learn to properly label embedding vectors from the latent space. We seek a metric that better rewards the classifier the more it is confident about correct predictions. Indeed, given our inference workflow, it is important to discern among predictions based on a confidence value: easy instances that are classified correctly in the local classifier should feature a high level of confidence, while a low confidence value should be assigned to predictions whose correctness is uncertain. This is crucial to decide between accepting the local prediction and paying the price to solicit the intervention of the remote model.

For this, we use a negative log-likelihood classification loss

$$\mathcal{L}_C = -\textstyle\sum_{x \in X} \sum_{c=1}^{C} \mathbb{1}_{\{c = l(x)\}} \log \gamma_c(z), \tag{4}$$

where $\gamma_c(z)$ is the prediction confidence, i.e., the early classifier assigns label c to the embedding $z \sim N(\mu(x), \sigma(x))$ with probability $p(c|z) = \gamma_c(z)$. Using such loss, the value $\max_c(\gamma_c(z))$ provides a good indicator of how confident the local classifier is about its prediction for embedding z.

D. Training Workflow

We minimize a weighted combination $\mathcal{L}$ of the three losses

$$\mathcal{L} = \alpha\mathcal{L}_M + \beta\mathcal{L}_R + \gamma\mathcal{L}_C, \tag{5}$$

where α, β, and γ are weights that we determine empirically (see Section VI). In order to minimize (5), we propose a training workflow where the two heads Head_1 and Head_2 are trained jointly together with the early classifier.

Since the very beginning of the process, the embeddings obtained through Head_1 from the training set are forwarded both to the Head_2 and to the early classifier. Based on the two outputs, we compute the loss (5) and all the three pieces of the model are updated jointly with SGD using backpropagation. Intuitively, this will affect the learning of the latent space in a way that takes into account the correctness of the local classifications, optimizing a trade-off between the goals of Head_2 and of the local classifier. Note that this joint training is the only way to properly express the regularization loss, as the parameters of the prior $p(z)$ depend from the early classifier.

Note that, not contributing to $\mathcal{L}$, the Tail is not affected by this training process. Indeed, by minimizing the mimic loss $\mathcal{L}_M$, Head_1 and Head_2 mimic the output of the original head, and for $\mathcal{L}_M = 0$ the Tail achieves the same accuracy as the original model. In general, the Tail weights may be preserved accounting for a small reduction in accuracy (as done in [33]). However, it is possible to further improve the accuracy of the remote model by fine-tuning Head_2 and Tail after the process above is complete. We do this by freezing Head_2 and minimizing the Cross Entropy between the Tail output and target labels in the training set.

V. Gaussian Mixture Classifier

The embeddings that we aim to classify locally are obtained early on in the feature extraction process. Accordingly, we do not expect a perfect structure of the latent space even after regularization; hence, traditional classification approaches (e.g., logistic) may fail to operate on such kind of data. Consistently with our Gaussian mixture assumption for the prior distribution, we propose an expressive classification model that estimates the probability of each label $c \in \{1, \dots, C\}$ based on a mixture of K normal distributions (Gaussian Mixture Classifier — GMC). We assign probabilities to each label through a weighted sum among all the components

$$\gamma_c = \frac{\sum_k \pi_{ck} N(z; \mu_{ck}, \Sigma_{ck})}{\sum_{c'} \sum_{k'} \pi_{c'k'} N(z; \mu_{c'k'}, \Sigma_{c'k'})}. \tag{6}$$

In practice, this is equivalent to directly learning the prior (2) that best fits the data. Our GMC architecture is inspired by [39]. We learn a group of parameters for each component c, k: *(i)* $\pi_{ck}^{(p)} \in \mathbb{R}$, that is the weight of component c, k in the mixture, *(ii)* $\mu_{ck}^{(p)} \in Z$, the mean vector of component c, k, and *(iii)* $L_{ck}^{(p)} \in \mathbb{R}^{d \times d}$, the Cholesky decomposition [40] of the covariance matrix (where $d = \dim(Z)$). Differently from [39], we do not limit our model to diagonal covariance matrices. However, as the covariance matrix is symmetric, we limit the number of parameters to $d(d+1)/2$, i.e., $L_{ck}^{(p)}$ is a lower-triangular matrix.

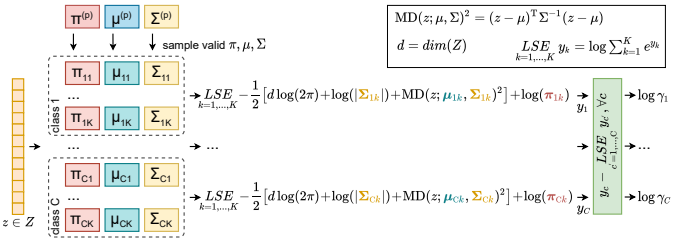


Fig. 5: Our GMC operates in log domain. First, valid π, μ and Σ are sampled from the trainable parameters. At inference time, we compute the log probabilities y_c for each class c, weighting the K components. The last block outputs the log probabilities weighted with respect to the other classes.

A Gaussian mixture has some constraints on its parameters. Hence, we use the trainable parameters $\pi_{ck}^{(p)}$, $\mu_{ck}^{(p)}$, and $L_{ck}^{(p)}$ (marked with the apex $\cdot^{(p)}$) to sample the actual parameters of the Gaussian mixture as follows:

1. *weights* π_{ck} — we ensure positivity and sum-to-one property by sampling $\pi_{ck} = e^{\pi_{ck}^{(p)}} / \sum_{c',k'} e^{\pi_{c'k'}^{(p)}}$;
2. *covariance matrix* Σ_{ck} — we sample symmetric positive-definite matrices $\Sigma_{ck} = L^{(p)\mathrm{T}} L^{(p)} + I$;
3. *mean* μ_{ck} — there are no requirements for the mean vectors, hence $\mu_{ck} = \mu_{ck}^{(p)}$.

The architecture of ReBEL's GMC is illustrated in Fig. 5. Using this early classifier, the regularization loss in (3) is obtained easily, as we can compute the KL–divergence between the distribution learned by the GMC (prior) and the output of Head_1 (posterior).

VI. Experimental Results

We evaluate ReBEL on ResNet50 with the CIFAR-100 dataset. Additionally, we also evaluate performance in unbalanced scenarios where different clients have local datasets drawn from different distributions. For local predictions, we use the proposed Gaussian Mixture Classifier (GMC) and compare it against other early classification approaches adopted by [3], [16], [36]. To evaluate the impact of the latent space regularization of ReBEL, we train the models both with and without the regularization loss from Eq. (3). In our experiments, we investigate the inference accuracy of the fraction of queries that are decided by the local classifier (i.e., those with sufficient prediction confidence). Last, we quantify the practical advantages of ReBEL over full remote computing offload and vanilla HND [33].

A. Experiment Setup

Model architecture. The reference model architecture for our experiments is ResNet50, a deep neural network with 53 conv. layers, where the first two groups of residual blocks (i.e., 24 conv. layers) are replaced by our variational head (4 conv. layers on Head_1, deployed device-side, and 4 conv. layers on Head_2, deployed server-side), with bottleneck size $dim(V) = 867$. With such a bottleneck, the embeddings' serialized footprint is 0.28 times the size of the original images. The rest of the architecture is left unchanged (Tail with

29 conv. layers, deployed server-side). We test our approach on the CIFAR-100 dataset, which features 60K image samples (50K for the training set, 10K for the validation set) of size $3 \times 32 \times 32$. We train a baseline bottleneck-injected model on CIFAR-100 using HND [33], reaching an accuracy of 76.4%.

Compute and network. We characterize two distinct execution nodes for the IoT device and for the server. Server-side, we run our experiments on an AWS EC2 instance equipped with a Tesla T4 GPU, where we deploy Head_2 and Tail. To represent the device, we run the Head_1 and the local classifier on a CPU (4 core Intel Xeon 8259CL), which is about 55 times slower than the GPU (a similar performance proportion as in [33]). We model the network between the IoT device and the server as an ideal link with 10 ms of propagation delay and evaluate a range of bandwidth settings from 1 to 10 Mbps.

Local classifiers. We implemented the Gaussian Mixture Classifier (GMC) proposed in Section V as a PyTorch module that can be trained with SGD and back-propagation. In our tests, we use K=8 components per class and add a random-projection module that reduces the dimensionality of the latent space from 867 to 434 using the algorithm from Achlioptas [41].[2] We evaluate also a more lightweight version of the GMC that only uses diagonal matrices. We compare the performance of GMC with three other approaches commonly used in literature for early classification:

1. *K–NN* — Assigns labels based on the *K* closest items in the latent space. As it requires querying the whole training set at inference time, it is computationally cumbersome. We use the FAISS[3] optimized implementation and empirically choose K=100. We use a confidence metric similar to [36].
2. *K–means* — Embeddings are divided into K clusters and, at inference time, labels are assigned based on the closest centroid (the label that was predominant in that cluster is assigned). As in [36], the confidence is measured as the share of the predominant label in each cluster. We set *K*=800, i.e., on average, 8 clusters per class.
3. *Linear w/ Softmax* — A linear transformation followed by Softmax activation, this classifier is typically used at the end of a DNN and is adopted in [16] for early classification.

We empirically set the loss function coefficients in (5), as α=1, β=0.001, and γ=0.1. Note that, when we use *K*–NN and *K*–means as local classifiers, we can not compute the regularization loss from Eq. (3). Therefore, we only evaluate these models without regularization. Also, note that the "Linear w/ Softmax" model is equivalent to a unimodal Gaussian mixture where all components share the same covariance matrix. Hence, we can compute the regularization loss sampling a prior through the relations described in [42, Appendix A].

Table I lists the relevant statistics for all the classifiers; as a reference, we also show the statistics for the original ResNet50 model and for the bottleneck-injected model, distinguishing between the device side (Head_1) and the server-side (Head_2 + Tail). We remark that the *K*-NN algorithm, despite the Faiss optimizations, has computational requirements of the same order as the original model, hence it is not practically suitable for early classification. However, the GMC classifier outperforms *K*-NN, even in its diagonal version.

TABLE I: Statistics of the models employed in our experiments. Floating point operations (FLOPs) were measured with the PAPI tool [43].

Model	FLOPs	# Parameters	Memory
ResNet50	196 M	25.6 M	67.7 MB
Head1	4.72 M	32.0 K	125 KB
Head2 + Tail	153 M	22.8 M	87.0 MB
Faiss K–NN	142 M	-	168 MB
Faiss K–means	2.25 M	-	2.96 MB
Linear	13.2 K	86.8 K	339 KB
GMC	19.6 M	75.9 M	291 MB
GMC (diag)	498 K	695 K	4.09 MB

B. Accuracy of local predictions

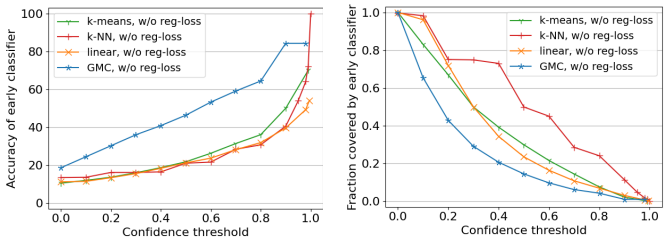
We first train the four early classification models omitting the regularization loss. Each model learns its own confidence metric in $[0, 1]$ that we use to decide whether we should accept the local predictions (easy queries, with confidence above a certain threshold) or further processing is needed (hard queries with low confidence). As shown in Fig. 6a, by setting higher threshold values, all the classifiers provide increased accuracy. This suggests that all of them successfully learn a meaningful confidence metric, i.e., higher-confidence predictions are more often correct than lower-confidence ones. Fig. 6b depicts the relationship between the chosen threshold and the fraction of queries covered by the local classifier (i.e., their prediction confidence is above the threshold). Note that as each classifier learns a different confidence metric, they feature different relationships between confidence threshold and coverage. For this reason, the threshold should be set differently for each model to cover the same fraction of queries locally. Since this prevents us from comparing the different approaches simply relying on the confidence threshold, in the following we will compare models relying instead on the fraction of queries covered by the early predictions. This allows us to clearly understand the performance of each model based on the fraction of queries it serves locally.

Fig. 7a compares local classifier accuracy based on the fraction of queries covered locally. Dashed lines show the impact of regularization on linear and GMC models[4]. Results show that our regularization process noticeably improves the performance of both classifiers: under an equal fraction of covered queries, GMC increases its accuracy by up to 14%.

[2]Note that this makes our GMC operate on a lower-dimensional space compared to Z, significantly reducing the number of parameters, but also introducing a mismatch between the dimensions of prior and posterior. Hence, to compute the KL–divergence we first apply the same dimensionality reduction to the posterior, then evaluate Eq. (3) in the lower-dimensional space.

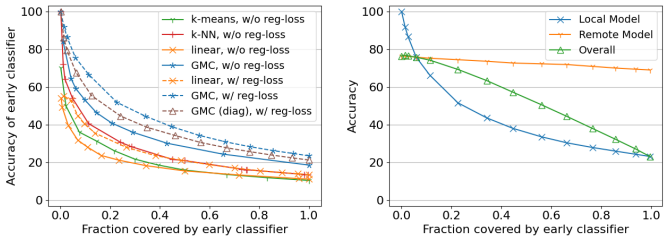
[3]Facebook AI Similarity Search (https://ai.facebook.com/tools/faiss/)

[4]Regularization also affects the accuracy of the remote model: we reached 75.2% when training with the linear classifier and 76.7% with the GMC. Remarkably, the latter is slightly higher than the base model accuracy (76.4%). This suggests bottleneck regularization might also increase remote classifier accuracy instead of simply trading it off for improved local predictions.



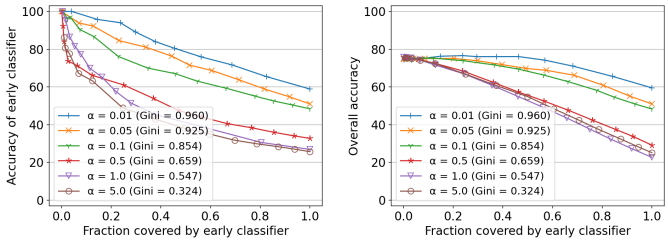
(a) Local accuracy. (b) Fraction of queries served locally.

Fig. 6: Confidence threshold impact on (a) local accuracy and (b) fraction of local predictions by early classifiers trained without regularization. Each classifier achieves a different confidence distribution for its local predictions.



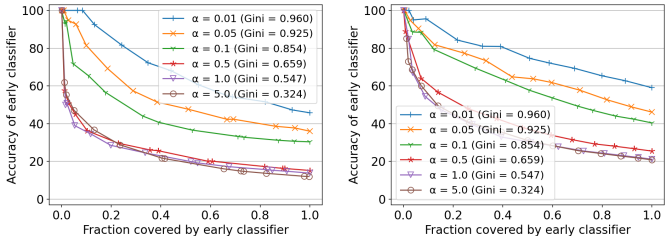
(a) Local accuracy. (b) Overall accuracy of ReBEL with GMC.

Fig. 7: (a) Accuracy of local classifiers vs. fraction of early predictions. Dashed lines show improvements introduced by regularization. (b) Local, Remote, and Overall accuracy of ReBEL using regularized GMC.



(a) Accuracy of GMC. (b) Overall accuracy of ReBEL with GMC.

Fig. 8: Accuracy of (a) GMC and (b) overall ReBEL approach on unbalanced datasets. Datasets are sampled from CIFAR-100 with a Dirichlet distribution: lower values of alpha lead to unbalanced datasets with high number of samples from few classes. The legend states the Gini index for each alpha.



(a) Accuracy of K-NN. (b) Accuracy of Diagonal GMC.

Fig. 9: Accuracy of (a) local K-NN classifier and (b) local diagonal GMC on unbalanced datasets. Datasets are sampled as in Fig. 8.

In general, the proposed GMC, even in its lightweight diagonal version, outperforms all other approaches, including *K*-NN, which is impractical due to its high complexity. When the fraction of queries covered locally is small (i.e., higher confidence thresholds are set), some local classifiers are able to provide levels of accuracy even higher than the remote classifier (i.e., 76.4%). In particular, GMC matches the remote model's accuracy when serving ≈10% of queries locally.

Fig. 7b outlines the accuracy of the local model (GMC with regularization) and of the remote model. We observe that, when most of the queries are served locally, the remote model only receives the hardest portion of the queries, thus it slightly reduces its accuracy. However, this trend is not marked for lower coverage values (below 0.4), where the remote classifier preserves its accuracy, suggesting that the local classifier specializes on a different subset of queries. The graph also shows the accuracy of the overall distributed model, i.e., accounting for both local and remote predictions. The results show that it is possible to cover 20% of the queries locally by accounting for less than a 5% drop in accuracy with respect to the remote classifier. When serving less than 10% of the queries locally, the remote accuracy is even slightly improved by local predictions.

Last, we evaluate the efficiency of our approach in terms of computational requirements. To this aim, we compare it with the approach recently proposed by Kumar et al. [36], where the output of intermediate layers of ResNet50 is directly exploited for early classification with *K*–means: the higher the number of used layers, the higher the fraction of queries that can be served locally. Table II compares the number of floating-point operations (FLOPs) needed to process different percentages of queries locally while preserving a certain fraction of the remote accuracy. In [36], authors provide the results when providing an overall accuracy that is 88.86% of the full one. To have a fair comparison, we do the same with respect to our remote model. For ReBEL, we report the results of GMC (both full and diagonal), also including the FLOPs required by Head_1. Results show that, to cover the same percentage of queries, our approach requires up to 14x less computation.

C. Accuracy on Unbalanced Datasets

The results above consider a scenario where both local and remote classifiers are trained over the same dataset. In a realistic IoT scenario, each local classifier may operate in different geographical areas, with access to different data subsets. In order to benefit from all the available information, the remote model would be trained through federated learning [44], while the training of local classifiers would take account only of local data. We show that this unbalanced situation can bring significant benefits in terms of accuracy of locally served queries, as our local classifiers can specialize on the particular data distribution in its geographical area.

To simulate this unbalanced scenario, we use a common approach in literature that generates random partitions of the main dataset sampling from a Dirichlet distribution [44]. The distribution assigns different probabilities to different groups of classes and features a parameter α: the more this parameter is close to zero, the more unbalanced the probabilities of sampling from different classes; the higher alpha, the more the distribution will be uniform among classes.

We choose some representative values of α and train our GMC model on the resulting datasets. Fig. 8a shows the

TABLE II: On-device FLOPs of ReBEL and Kumar [36] for different fractions of early predictions while delivering 88.86% of the original accuracy. ReBEL-GMC diag. and full serve respectively 22% and 26% of queries locally.

Served locally [%]	5%	22%	26%
Kumar et al. [36]	30 M (FLOPs)	74 M	92 M
ReBEL-GMC (diag|full)	—	5.2 M	24 M

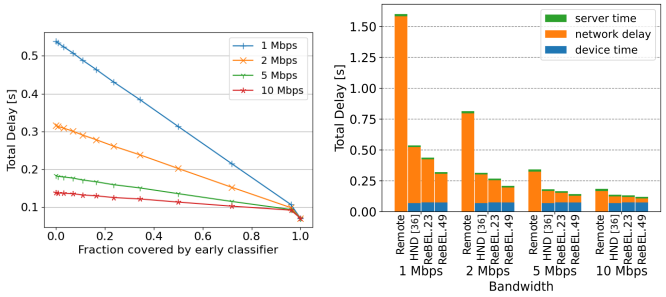
accuracy of the local predictions varying the fraction of queries covered locally. For every value of α, we provide the Gini index as an indicator of how unbalanced each dataset is in terms of samples per class. The figure shows that when the dataset is highly unbalanced, the local classifier performs remarkably well: for α=0.01 (Gini is 0.96, most samples belong to 5 classes only), we can obtain better accuracy than the remote model while serving more than half of the queries locally. As expected, by increasing α, the datasets become more balanced and this advantage disappears, hence results match those in Fig. 7. As we observe from Fig. 8b, for unbalanced datasets the usage of a local classifier may even improve the overall accuracy compared to serving all the queries remotely. In particular, for α=0.01, when 46% of queries are served locally, the overall accuracy gains additional 1.8% points compared with only using the remote classifier. This remarkable result suggests that, aside from reducing the inference time and the network load (results that we discuss in the remainder of this section), in many IoT use cases our approach may bring advantages also in overall accuracy, as the local model can specialize on local data.

Last, Fig. 9 compares, varying alpha, the performance of K–NN (that is the competitor with the highest accuracy) and our GMC model in its lightweight diagonal version. Results show that GMC outperforms K–NN even in its diagonal version, providing almost the same accuracy values as its full version, despite requiring much less computational power.

D. Inference Time and Network Load

In this set of experiments, we evaluate the benefits of our bottleneck-classification approach in terms of inference time reduction. Overall, we reduce the average inference time due to two factors: *(i)* the presence of a bottleneck between the device and the remote server, i.e., smaller embeddings are sent to the server compared with the original images (same as HND [33]), thus requiring less transmission time; *(ii)* a fraction of the queries is served locally, hence even fewer data should be sent to the server compared to HND. For this set of experiments, we used the Gaussian Mixture classifier with diagonal covariance matrixes and we measure the delay for batches of 64 queries.

Fig. 10a shows the average end-to-end delay for different network conditions (available per-device bandwidth), varying the coverage of early predictions. Of course, the overall delay decreases the more queries are served locally. Three factors contribute to the total delay: *(i)* processing time on the device for feature extraction by Head_1 and early classification (device time); *(ii)* transmission time of intermediate embeddings toward the server, plus RTT propagation delay (network time); *(iii)* processing time on the server. We neglect



(a) Total inference delay of ReBEL. (b) Server, network, and local delay.

Fig. 10: Overall prediction delay on batches of 64 samples for different network conditions. On the left, total delay of ReBEL varying the fraction of local predictions. On the right, decomposed delay (network, local, and remote computation) comparing different approaches (For ReBEL, we use diagonal GMC and show results for two different fractions of local predictions).

the extra transmission time for sending back the prediction results to the device, as their size is a handful of bytes. We select two different coverage values (0.23 and 0.49) and analyze more in detail the three delay components (Fig. 10b). The figure compares ReBEL both with traditional remote-only computation (using the original ResNet50 model and sending the full images towards the network) and vanilla Head Network Distillation (as in [33]). We see that our approach introduces a significant delay reduction when the available bandwidth is in the order of units of Mbps, which is a common scenario in IoT networks. When the device has access to larger bandwidth values the network time becomes comparable to the local computation time and the advantage introduced by our approach becomes smaller.

Finally, to provide an example of the practical gains of ReBEL in a realistic use case, we shall consider an IoT network with 100 devices, each generating 25 frames per second that need to be classified. The overall network footprint of the raw image transmission in a full remote offload scenario would be 61.44 Mbps. Bottleneck injection via HND [33] would reduce network load to 17.34 Mbps. Introducing early labeling at the bottleneck would bring the required throughput to 13.35 and 8.84 Mbps, for coverage values 0.23 and 0.49, respectively. The benefits of ReBEL in terms of network load are thus practically significant.

VII. Conclusions

This paper proposes ReBEL, a novel dynamic offload approach to split a DNN model computation between a local resource-constrained device and a remote server. Inspired by the work on Head Network Distillation (HND), we inject a bottleneck at the initial stages of a target pre-trained DNN model and enhance it with early exit capabilities. A key element of ReBEL is the design of an efficient bottleneck embedding based on the inclusion of two new terms into the training loss function: a regularization loss, based on variational inference modeling, and an early classification loss, with the goal of shaping the bottleneck contents for successful local classification.

We evaluated the behavior of a minimal on-device head network (4.72 MFLOPs, 32K parameters, 125 KB of mem-

ory) for local and overall inference accuracy on a CIFAR-100 image classification task. Our results demonstrate that a local classifier based on Gaussian Mixtures far outperforms other common early exit classifiers, such as linear, *K*–NN, or *K*–means, when used on a regularized early bottleneck output after a minimal amount of on-device processing. In our experiments, we also assess the advantages of specialized training for achieving higher accuracy on non-uniform input class distributions and highlight the expected gains in inference time and reduced network load. We believe ReBEL contributes an advanced split computing architecture that can efficiently bestow full-fledged AI capabilities onto constrained IoT devices with network access.

ACKNOWLEDGMENT

This work was funded by a joint lab contract between Inria and Nokia Bell Labs (ADR "Rethinking the Network").

REFERENCES

[1] Y. Matsubara *et al.*, "Split computing and early exiting for deep learning applications: Survey and research challenges," *arXiv preprint arXiv:2103.04505*, 2021.

[2] K. He *et al.*, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[3] C. Lo *et al.*, "A dynamic deep neural network design for efficient workload allocation in edge computing," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 273–280.

[4] S. Teerapittayanon *et al.*, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[5] Y. Matsubara *et al.*, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Comput. Surv.*, mar 2022. [Online]. Available: https://doi.org/10.1145/3527155

[6] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[7] G. Li *et al.*, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *Int'l Conference on Artificial Neural Networks*. Springer, 2018, pp. 402–411.

[8] A. E. Eshratifar *et al.*, "Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, 2019.

[9] H. Li *et al.*, "Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *24th IEEE Int'l Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 671–678.

[10] H.-J. Jeong *et al.*, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM symposium on cloud computing*, 2018, pp. 401–411.

[11] L. Lockhart *et al.*, "Scission: Context-aware and performance-driven edge-based distributed deep neural networks," *CoRR*, 2020.

[12] K.-J. Hsu *et al.*, "Couper: Dnn model slicing for visual analytics containers at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 179–194.

[13] Z. Zhao *et al.*, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[14] L. Zhou *et al.*, "Distributing deep neural networks with containerized partitions at the edge," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[15] S. Leroux *et al.*, "Resource-constrained classification using a cascade of neural network layers," in *2015 International Joint Conference on Neural Networks (IJCNN)*, 2015, pp. 1–7.

[16] P. Panda *et al.*, "Energy-efficient and improved image recognition with conditional deep learning," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, Feb. 2017.

[17] Y. Kaya *et al.*, "Shallow-Deep Networks: Understanding and mitigating network overthinking," in *Proceedings of the 2019 International Conference on Machine Learning (ICML)*, Long Beach, CA, Jun 2019.

[18] H. Li *et al.*, "Improved techniques for training adaptive deep networks," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE ComSoc, nov 2019, pp. 1891–1900.

[19] H. Hu *et al.*, "Learning anytime predictions in neural networks via adaptive loss balancing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3812–3821.

[20] K. Neshatpour *et al.*, "Exploiting energy-accuracy trade-off through contextual awareness in multi-stage convolutional neural networks," in *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019, pp. 265–270.

[21] M. Wang *et al.*, "Dynexit: A dynamic early-exit strategy for deep residual networks," in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2019, pp. 178–183.

[22] S. Teerapittayanon *et al.*, "Distributed deep neural networks over the cloud, the edge and end devices," in *37th IEEE Int'l Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[23] L. Li *et al.*, "Deep learning for smart industry: Efficient manufacture inspection system with fog computing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4665–4673, 2018.

[24] E. Li *et al.*, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[25] L. Zeng *et al.*, "Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things," *IEEE Network*, vol. 33, no. 5, pp. 96–103, 2019.

[26] Y. Wang *et al.*, "Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 623–633, 2020.

[27] D. Hu *et al.*, "Fast and accurate streaming CNN inference via communication compression on the edge," in *5th IEEE/ACM Int'l Conference on IoT Design and Implementation (IoTDI)*, 2020, pp. 157–163.

[28] A. E. Eshratifar *et al.*, "Bottlenet: A deep learning architecture for intelligent mobile cloud computing services," in *2019 IEEE/ACM Int'l Symposium on Low Power Electronics and Design (ISLPED)*, 2019.

[29] J. Shao *et al.*, "Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems," in *2020 IEEE Int'l Conference on Communications Workshops (ICC Workshops)*, 2020.

[30] Y. Matsubara *et al.*, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proc. of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019, pp. 21–26.

[31] G. Hinton *et al.*, "Distilling the knowledge in a neural network," in *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[32] Y. Matsubara *et al.*, "Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing," 2022.

[33] ——, "Head network distillation: Splitting distilled deep neural networks for resource-constrained edge computing systems," *IEEE Access*, vol. 8, pp. 212 177–212 193, 2020.

[34] D. P. Kingma *et al.*, "An introduction to variational autoencoders," *arXiv preprint arXiv:1906.02691*, 2019.

[35] J. Ballé *et al.*, "Density modeling of images using a generalized normalization transformation," *arXiv preprint arXiv:1511.06281*, 2015.

[36] A. Kumar *et al.*, "Accelerating deep learning inference via freezing," in *11th USENIX Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[37] G. Huang *et al.*, "Multi-scale dense networks for resource efficient image classification," *arXiv preprint arXiv:1703.09844*, 2017.

[38] S. Kullback *et al.*, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[39] E. Variani *et al.*, "A gaussian mixture model layer jointly optimized with discriminative features within a deep neural network architecture," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 4270–4274.

[40] N. J. Higham, "Analysis of the cholesky decomposition of a semi-definite matrix," in *Reliable Numerical Computation*. Oxford University Press, Oxford, UK, 1990, pp. 161–185.

[41] D. Achlioptas, "Database-friendly random projections," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2001, pp. 274–281.

[42] H. Hayashi *et al.*, "A discriminative gaussian mixture model with sparsity," *arXiv preprint arXiv:1911.06028*, 2019.

[43] D. Terpstra *et al.*, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[44] H. Wang *et al.*, "Federated learning with matched averaging," *arXiv preprint arXiv:2002.06440*, 2020.