2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

Eugene: Towards Deep Intelligence as a Service

Shuochao Yao*†, Yifan Hao*†, Yiran Zhao†, Ailing Piao‡, Huajie Shao†, Dongxin Liu†,
Shengzhong Liu†, Shaohan Hu§, Dulanga Weerakoon¶, Kasthuri Jayarajah¶, Archan Misra¶, Tarek Abdelzaher†
University of Illinois at Urbana-Champaign†, University of Washington at Seattle‡, IBM Research§,
Singapore Management University¶
Email: {syao9, yifanh5, zhao97}@illinois.edu, alpiao@uw.edu,
{hshao5,dongxin3,sl29}@illinois.edu, shaohan.hu@ibm.com, archanm@smu.edu.sg, zaher@illinois.edu

***Abstract*—The paper discusses an emerging suite of machine intelligence services that are of increasing importance in the highly instrumented world of the Internet of Things (IoT). The suite, called *Eugene*[1], would offer a form of intelligent behavior (based on deep neural networks) to otherwise simple embedded devices; the clients of the service. These devices would benefit from service resources to learn from data and to perform intelligent inference, classification, prediction, and estimation tasks that they are too limited to carry out on their own. The paper discusses the taxonomy of such services and the state of implementation, as well as the various challenges entailed, including scheduling, caching (of intelligent functions), and cooperative learning.**

I. Introduction

We aim to enable ubiquitous intelligence in a future world of connected sensing and computing devices, seamlessly embedded in our surroundings. We propose to do so using a service, tentatively called *Eugene*, that endows everyday objects with the appearance of human-like behavior and encyclopedic knowledge. The goal is to revolutionize our interactions with the physical world the way the Internet revolutionized out interactions with each other. In *Eugene*'s world, embedded IoT devices (or "things") will be capable of human-like interactions with their environment, including speech recognition, vision, and gesture understanding. These capabilities will bring about such features as verbal device control, (soft) user authentication, and gesture-based human machine communication.

Eugene would accomplish the above goals by allowing the offloading of machine intelligence tasks. Indeed, the disparity between the resource-constrained nature of embedded IoT devices and the computational needs of the aforementioned interactions suggests that data processing will be increasingly offloaded to external servers. Today, precursors of such services include speech recognition for home controllers (e.g., Amazon Echo) and language translation for mobile phones, both done partly in the cloud. With the increasing popularity of *edge computing*, external servers will likely move closer to the clients, and some functionality will be "cached" on the local device. A business, such as a management service for a shopping mall, for example, might host its own edge servers to satisfy the needs of local IoT devices. These devices might include mall surveillance cameras, smart fitting rooms that suggest better-fitting items to customers, audio-based chatbots that offer directory assistance, and indeed customers' own phones (that run the appropriate app). Inference models that support certain interactions might be downloaded (after simplification to reduce size) to the individual IoT devices engaged in those interactions as a form of "caching". Caching appropriately trained neural network models will offer portable intelligence for heterogenous devices that aim to run limited intelligent inference functions locally.

We argue for realizing *Eugene* using deep neural networks as the instrument of machine intelligence. This choice is motivated by the emergence of deep learning as the state-of-the-art computational intelligence solution for a large spectrum of IoT applications [1]. Besides breakthroughs in processing images and speech using deep learning techniques [2], [3], specific neural network structures have been designed to fuse multiple sensing modalities and extract temporal relationships [4]. The increasing number of studies on applying deep learning in the area of cyber-physical systems (CPS) and IoT [4]–[7] make it a prime candidate for realizing the intelligent capabilities of *Eugene*.

The rest of this paper is organized as follows. Section II introduces the key functional requirements of *Eugene*. Challenges in the underlying system support, namely, back-end scheduling, are presented in Section III, together with a preliminary evaluation. We discuss challenges in supporting cooperative intelligence in Section IV. Finally, we conclude in Section V, and outline other possible future work.

II. Core Service Requirements

Eugene will offload from IoT devices the training and/or execution of machine learning algorithms, such as classifiers or predictors, to do a myriad of common estimation and recognition tasks based on device data such as visual inputs, speech, or gestures. It is possible to concieve of *Eugene* as a *virtual machine for artificial intelligence*. Like other virtual machines (e.g., Java and Python), it would allow the expression of processing tasks in some efficient intermediate form. This form, we argue, is the neural network model. The model specifies network topology and edge weights, as well as other hyperparameters such as the type of activation functions used. With those parameters, it becomes possible to implement

*Equal contribution

[1]Named after Eugene Goostman, an AI simulation of a boy, claimed to be the first machine passing the Turning test in a controvesial result of a competition in 2014.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00162

inference algorithms (specified by the model) that perform classification, prediction, or estimation functions.

Clients would ask the service to (i) generate deep neural network models (from client-supplied training data), (ii) help with (automatic) labeling of data sets, and (iii) perform model reduction (if needed for caching). Generated models might be executed as appropriate on the server, client, or any device that supports the "virtual machine". System support is needed on servers to enable efficient scheduling of inference tasks (that execute the computed models on incoming client-supplied data in real-time). A scheduler might maximize a suitably defined notion of utility to improve quality of inference results. Auxiliary functions are needed such as profiling. They will allow enhanced (neural network) model parameterization to improve accuracy and/or cost.

The feasibility of developing *Eugene* as such a general-purpose service is attributed primarily to the general-purpose nature of deep learning itself, making *Eugene* largely autonomic and configuration-free. If the service requires lengthy per-application engineering and customization, it will lose much of its appeal. In this regard, deep learning frameworks have at least two key advantages over alternative solutions:

- Arguably, in many scenarios, one can use laws of physics to derive the needed inference results from sensor data. For example, in a location estimation task, one can double-integrate inputs that comprise accelerometer data to obtain velocity and position. The problem with such approaches is two-fold. First, they require that application-specific models of underlying physical phenomena be developed and given to the service. Second, they rely on understanding accurate models of noise. Most estimators make assumptions on the statistical distribution of noise offering accurate results only when such assumptions are satisfied. In a complex environment, noise is hard to model. It may be non-linear, non-additive, correlated, and biased. Recent results in deep learning demonstrate that the network can learn very complex nonlinear relations, allowing better extraction of signals from noise (even when the two are intertwined in a complex nonlinear fashion) [4]. Best of all, such extraction is fully automated, thus requiring no human intervention or expertise.
- Furthermore, unlike other machine learning approaches that rely on the design of clever input features (to support the intended estimation or classification tasks), deep learning has the advantage of being able to ingest raw data directly and automatically compose relevant features by adjusting link weights. Hence, less human effort is consumed in feature engineering.

In a world dominated by data and computing devices, saving human cognitive bandwidth by employing a machine is a great trade-off. With that in-mind, we set forth to describe what the *Eugene* general-purpose machine intelligence service suite should be able to do.

A. Training and Data Labeling

Eugene will facilitate *learning* from data collected by the embedded devices. These services will execute on the back-end to produce the trained neural networks necessary for various inference and estimation tasks. The most basic service is to ingest labeled raw data from clients and train the eventual neural network model on the server. Since it is expensive to label a lot of data manually, another service would be to assist with *automatic* labeling. Below we describe the underlying challenges and possible solutions in more detail.

Training: The first challenge in implementing deep intelligence as a service lies in training the neural network to support the application of choice. In many cases, IoT devices will have already collected large amounts of sensory data (such as video footage from security cameras). Often, labels are available retrospectively (such as instances of various security breaches caught on camera). This offers opportunities for training the system to identify (and alert to) similar instances in the future. The feasibility of such a service was recently discussed in DeepSense [4], a general-purpose learning framework for sensor fusion systems. It integrates convolutional neural networks (CNNs) and recurrent neural networks (RNNs) to extract spatio-temporal features of input signals. Sensory data are aligned and divided into time intervals for processing. For each interval, DeepSense first applies an individual CNN to each sensor data stream, encoding relevant local features. A (global) CNN is then applied to the respective outputs to model interactions among multiple sensors for effective sensor fusion. Next, an RNN is applied to extract temporal trends. Intelligent IoT applications will generally need two important functions: estimation and classification (depending on whether the sought results are continuous or categorical, respectively). Hence, at the last stage, either an affine transformation or a softmax output is used by DeepSense, depending on whether the output is an estimation or a classification result. Accordingly, it becomes possible to perform complex multi-sensor fusion tasks for purposes of estimation or classification from time-series data. The detailed mathematical formulation of the DeepSense learning algorithm can be found in a related paper [4].

Labeling: A general disadvantage of deep learning methods lies in the need for large amounts of *labelled* data. To learn well from empirical measurements, the neural network must be given a sufficient number of labelled examples from which network parameters are to be estimated. Since the number of parameters is large, so is the required number of labelled examples. In order to make deep learning services practical, a key challenge is thus to reduce the need for labeled data. *Eugene* could address this challenge by employing a recently proposed approach that uses Generative Adversarial Networks (GAN) to learn from mostly unlabeled data [8]. Unlabeled data carries information on the structure of the input space. By overlaying it with labeled data, one can better observe the emergence of input data clusters corresponding to different labels. A small number of labeled points within a cluster can thus inform the labeling of the remaining points. Using this

intuition, the GAN learns by playing a game of progressive refinement of both the dimensions in which points are virtually clustered and the rules for cluster separation. In this game, one entity proposes labels for unlabeled samples, whereas another tries to distinguish the resulting labeled samples from the original labeled ones. As the game procedes, both entities learn from each other ultimately producing labels that are hard to falsify. Empirical results show that these eventual artificially produced labels (for originally unlabaled data) help improve accuracy of learning applications almost as much as the ground-truth labels themselves [8]. The approach significantly reduces reliance of the learning service on availability of large amounts of labeled data, allowing the exploitation of more easily attainable unlabeled data instead. The limitations of this approach remain to be investigated, but preliminary evidence suggests that it is effective at circumventing the lack of sufficient data labels.

B. Model Reduction and Caching

Once trained, deep neural networks can be used to perform complex estimation, prediction, detection, or identification/classification tasks. Typical networks produced by deep learning techniques are very large. They may include several hundreds of layers, each composed of possibly thousands of nodes. As such, they need to execute on appropriately well-resourced machines, resulting in communication between end-devices (e.g., sensors making new observations) and well-resourced back-ends every time the device needs to run the service on a new data item. In environments where the communication bandwidth of the end-device is not plentiful, it is advantageous to execute some inference tasks locally. This need calls for reducing the relevant neural network models to a footprint that fits the end device. Hence, a model reduction service is needed.

The feasibility of an efficient neural network model reduction service is attributed to two observations. First, it is often the case that phenomena observed by sensors evolve over lower-dimensional manifolds. In this case, the large neural network is an overkill and compression is possible. Second, in many applications, the most frequent inputs to a device comprise only a very small fraction of the much larger potential input space. For example, in a service where users typically give yes/no answers, recognizing responses such as "yes" and "no" versus neither (referring to all other utternaces besides these two) should be easier than distinguishing all possible spoken words. In this scenario, neural networks produced by deep learning methods can be reduced in size without significant loss of accuracy in the common case. Much like caching, a reduced network model can run locally on the resource-limited embedded device to handle common inputs (e.g., to recognize "yes" and "no" in the above example). The identification of an uncommon occurrence (e.g., the occurrence of other words) is viewed as a cache miss that triggers full network execution on the server.

Several attempts were made to simplify deep neural networks after they have been trained. Commonly, a compression service removes edges that have low weights. The removal produces a sparse matrix (to represent the neural network), where most of the cells are zeros. The sparsity of the matrix allows for reductions in storage and computation time. Unfortunately, prior work has shown that these reductions do not scale proportionally to the fraction of zero entries in the sparse matrix [5]. This is because sparse matrix algebra is not as efficient as dense matrix algebra. Hence, as the matrix becomes sparse, additional overhead is introduced to take advatage of sparsity (compared to when it was dense), thereby offsetting some of the savings. A promissing solution for a model reduction service is one that removes nodes instead of edges in the neural network to fix the above sparse matrix problem. Removal of entire nodes from the neural network is equivalent to removal of entire rows/columns from the corresponding matrix. This produces a new matrix that is also dense, but that has smaller dimensions. The approach was shown to be significantly more effective at reducing resource consumption without degrading quality [5]. The resulting compact neural network models are therefore suitable for execution on resource-limited nodes.

To automate caching, *Eugene* must decide on what constitutes frequent inference tasks. The inference models (i.e., neural networks) pertaining to those specific tasks can then be reduced and cached. For example, in a vision-based item identification system executed in a smart refrigerator, the most common items entered might end up being beer and pop bottles. Recognizing that the most common classification results point to those specific items, *Eugene* (running on the server) may retrain a neural network with only those items as positive examples, compress the result, and download the compressed model to the device. Several interesting questions arise in implementing this mechanism. For example, when exactly should the system decide that an item or set of items are frequent? How small or large should the set of items be to make it worth developing a reduced model for? How to automatically adapt answers to the above two questions according to the capability of the local device, and the bandwith of its communication link? Finally, when should the cached model be removed from the device? These questions are a topic of future work.

C. Execution Profiling

On the server side, execution efficiency considerations suggest the need to understand the relation between neural network structure and execution overhead. Prior work has shown that simply counting the number of neural network parameters and/or the total FLOPs involved in processing does not lead to good estimates of execution time because the relation between these predictors and execution time is highly non-linear [9]. Table I (reproduced from [9]) shows that networks with the same number of FLOPs (e.g., CNN1 and CNN2) can differ significantly in execution time. In fact, networks with fewer FLOPs can take longer to execute (e.g., CNN3 compared to CNN4).

Understanding the causes of nonlinear relations between network parameter settings and the resulting execution time,

TABLE I: Execution time of convolutional layers with 3×3 kernel size, stride 1, same padding, and 224×224 input image size on the Nexus 5 phone.

	in_channel	out_channel	FLOPs	Time (ms)
CNN1	8	32	452.4 M	114.9
CNN2	32	8	452.4 M	300.2
CNN3	66	32	3732.3 M	908.3
CNN4	43	64	4863.3 M	751.7

energy, and memory consumption is thus key to developing efficient deep learning service implementations. One may leverage recent work [9] that addressed the above challenge by implementing an automated profiling system that breaks execution models into piece-wise linear regions, and uses regression over the (automatically identified) relevant neural network parameters within each region to develop a predictive model of execution time in that region. A similar approach can be developed for modeling/minimizing energy or memory consumption. Such a profiling tool would optimize performance on the server side (as it will typically not have access to profiling results on the client). For example, leveraging the identified nonlinear behavior, it might become possible to *increase neural network size and accuracy* while at the same time *reduce its execution overhead* (as illustrated by comparing CNN4 to CNN3 in Table I).

D. Result Quality Estimation

Another important challenge in realizing intelligence as a service is to assess the quality of inference results produced by learning models. To support mission-critical applications, the service must offer principled uncertainty estimates that faithfully reflect the correctness of its predictions. Methods are needed that provide accurate uncertainty estimates in results obtained from deep learning models. Moreover, the uncertainty estimation must be resource efficient.

Recently, a well-calibrated and efficient uncertainty estimation algorithm was proposed for multi-sensor data fusion, called RDeepSense [6] (as an extension of DeepSense [4]). It emits a distribution estimate instead of a point estimate at the output layer. Intuitively speaking, the algorithm models node outputs with random variables and estimates their distribution parameters. Estimation of the mean of the random variable is what traditional learning does. Estimation of the variance, however, is what yields confidence in results. A smaller estimated variance corresponds to a higher confidence in the computed mean.

Interestingly, the estimation of the mean and the estimation of the variance are interrelated. Typically, the estimator jointly determines both by minimizing some error function. The choice of that function has an important effect on estimation accuracy of the two parameters. Specifically, using common error functions, such as the mean square error, was shown underestimate the uncertainty. This is so because such an estimator predicts a very accurate mean value. If the mean value is estimated well, the variance observed around that mean on training data is small and may thus underestimate variance encountered later during testing. In contrast, when using a nonlinear error function, such as the negative log-likelihood, the estimated mean is often biased (because the nonlinearity penalizes erring on one side more than erring on the other, causing the estimated mean to drift towards the heavily penalized side). The biased (i.e., incorrect) mean estimate results in increased measured variance around the mean, leading to an artificially inflated uncertainty estimate.

One can exploit the above intuition to arrive at an estimate of variance that neither underestimates nor overestimates the true value. The idea is to use a weighted sum of the above two error functions (namely, mean square error and negative log-likelihood) as the combined loss function [8]. The weights are adjusted (calibrated) such that the underestimation and overestimation roughly cancel out. RDeepSense was shown to generate very good uncertainty estimates that allow defining accurate confidence intervals for outputs of the deep learner.

The ability to compute confidence in deep learning results offers another interesting resource optimization possibility. Namely, one may structure a deep neural network into stages, each consisting of several layers, and compute confidence in (intermediate) results after each stage. Once a high-enough confidence is reported, it becomes possible to skip the execution of the remaining stages. For example, consider a deep neural network whose job is to identify the presence of humans in a landscape. The presence of humans may be easier to identify in some images than others. Consequently, it could be that fewer stages need to be executed for some images to reach an acceptable level of confidence in results. We return to this topic again when we describe challenges in back-end scheduling that aims to maximize total utility of the service.

E. Run-time Inference

It remains to describe the challenges in implementing the run-time inference service itself. The goal is to perform inference with a required degree of quality. The service would accept data from end devices that choose to offload inference processing to the server, and return inference results together with a confidence estimate. An important design consideration is scalability, which calls for execution efficiency. Recent studies on deep learning have shown that improvements in result accuracy diminish with increased depth of the neural network [2]. Hence, efficiency considerations suggest that once the desired quality is achieved, the service should refrain from executing additional layers.

One idea would be to schedule inference tasks in a way that optimizes total utility. The resulting overall run-time inference architecture is described in Figure 1. As shown in Figure 1, the deep neural network is separated into multiple layers. These layers are grouped into a small number of stages (of multiple layers each). At the end of each stage, a thin softmax function layer is attached to compute a classification at selected internal layers, as well as confidence in such classification. The scheduler determines how many stages to execute to avoid diminishing returns. More on the scheduling challenge is discussed below.

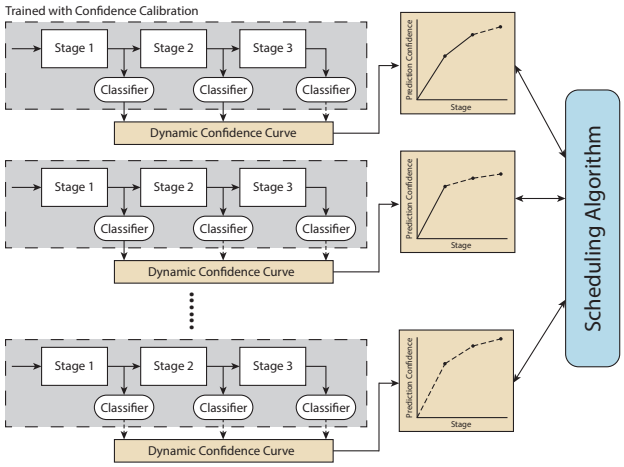


Fig. 1: The overview of Deep Intelligence as a Service.

III. System Support and Service Utility Maximization

While the previous section described service-level challenges in implementing basic intelligence as a service, this section describes challenges in the underlying system support. Specifically, we focus on scheduling on the server. The challenge in scheduling comes from the fact that the level of difficulty of inference tasks in deep learning engines is heavily influenced by the input data. For example, identifying a face in a picture could be a very easy or a very difficult task, depending on the picture. As alluded to above, one therefore needs to customize the executed neural network depth to each data item received. At some point, execution of additional layers reaches a point of diminishing returns and the priority of such execution should be reduced. This customization needs coordination between the service and the underlying scheduler.

We argue for implementing a utility-maximizing scheduler for *Eugene*'s inference tasks to improve the cost and scalability of the service. The goal of the scheduler would indeed be to choose the best inference depth for each task such that the overall utility is maximized. For a proof of concept of such a scheduler, we implemented a greedy algorithm that picks the next task stage to execute such that the maximum increase in accrued utility is achieved. The algorithm starts from an empty set. In each step, the algorithm picks a stage of a task with the maximum differential utility (where utility in our implementation is set equal to the estimated confidence in results). This selected stage is added to the future timeline. A lookahead parameter, k, specifies how many items will be added to the timeline before the scheduler quits. When the timeline has been executed, the algorithm restarts again with the most recent utility estimates given the current partial execution, and selects the next k stages.

The scheduling framework is implemented in user space. Implementing the scheduler in *user space* solves two key concerns. First, it does not require changes to the operating system, making it portable to more platforms. Second, it enables us to integrate the scheduler with widely deployed deep learning libraries. Specifically, we integrate it with TensorFlow [10]. For historic reasons, we call the *Eugene* scheduler RTDeepIoT.

To prevent unbounded delays, the scheduler can accept a latency constraint that specifies how long a given task can stay in the system before its execution needs to be finished. A daemon process monitors the elapsed time for each task. If the elapsed time for a task exceeds the maximum latency constraint, the daemon process will send a signal to stop the current computation. The process is returned to the pool and is made available to handle new requests. No utility is accrued for tasks that are not completed. The interaction between the scheduler and the service framework is thus as follows:

1) Input data arrive with requests for inference. They are assigned to one of a pool of waiting processes. The utility of executing the next stage is computed by the service.
2) The scheduler updates its estimate of utility of future stages and recomputes the set of stages to execute next.
3) When a stage is finished, the process sends the updated confidence value in results of subsequent stages to the scheduler.
4) If the process finishes all the stages of the current inference task, it goes back to the pool and waits for new assignments.
5) If the process cannot finish by the deadline, it will be interrupted by the daemon process, and forced to return to the pool.

Two challenges arise in implementing the above scheduler:

- *Confidence estimation:* How to estimate confidence in neural network outputs at intermediate layers?
- *Dynamic utility curve updates:* How to adapt the utility curve dynamically over time?

We discuss these two functions in more detail in the following subsections.

A. The Utility Metric: Confidence

Consider a classification problem as a running example of an inference task performed by a deep neural network. The output of a neural network classifier is a vector of probabilities, where the largest probability is called the classification confidence. Ideally, a well-calibrated classification confidence should be equal to the actual likelihood of classification correctness. Unfortunately, most deep learning systems are not well-calibrated in that sense. With the growing capability and advances in deep learning, although classification accuracy has greatly improved, the classification confidence is not as accurate [11].

The calibration of confidence can be visually represented by the reliability diagram [12]. As shown in Figure 2, the diagram plots expected classification accuracy as a function of confidence. If the neural network is perfect, then the diagram should plot the identity function. Any deviation from a perfect diagonal represents miscalibration.

In order to represent the degree of miscalibration with a scalar that summarizes statistics of calibration, we introduce

the metric, Expected Calibration Error (ECE) [13]. First, we group classification results into M bins with equal-width $1/M$. We denote $\mathcal{S}_m$ as the set of samples whose classification confidence falls into the interval $((m-1)/M, m/M]$. Then, we can define the average accuracy of $\mathcal{S}_m$ as:

$$acc(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} \mathbb{1}(\hat{y}_i = y_i), \tag{1}$$

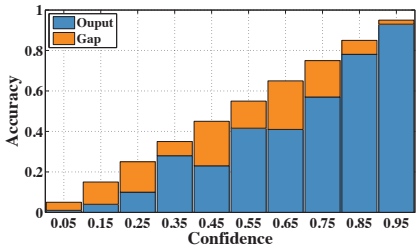
where $\hat{y}_i$ and y_i are the predicted and true label of sample $\mathbf{S}_i$. Next, we define the average confidence of $\mathcal{S}_m$ as:

$$conf(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} p_i, \tag{2}$$

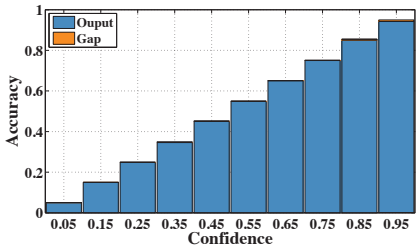
where p_i is the classification confidence of sample $\mathbf{S}_i$. The ECE metric is defined as the weighted average of the difference between average accuracy and confidence in M bins.

$$ECE = \sum_{m=1}^{M} \frac{|\mathcal{S}_m|}{m} \Big| acc(\mathcal{S}_m) - conf(\mathcal{S}_m) \Big|. \tag{3}$$

Accurate confidence estimation has drawn growing attention in recent studies [6], [14], [15]. However, existing efforts tend to either underestimate or overestimate the confidence [14], [15]. We denote by $\mathcal{S}$ the set of all samples. When $acc(\mathcal{S}) < conf(\mathcal{S})$, the neural network tends to underestimate the classification results. When $acc(\mathcal{S}) > conf(\mathcal{S})$, the neural network tends to overestimate. The target is to make $acc(\mathcal{S}) \approx conf(\mathcal{S})$ and $ECE \to 0$.



(a) Without confidence calibration



(b) With the entropy-based calibration.

Fig. 2: The reliability diagrams of ResNet on CIFAR-10.

A natural metric to control the classification confidence is entropy, $H(\mathbf{p}_i)$, where $\mathbf{p}_i$ is the vector of confidences over all targeted classes. Therefore, we propose a simple entropy-based regularization method for confidence calibration with fine-tuning. We reformulate the loss function of the fine-tuning process as:

$$\mathcal{L} = CE(\mathbf{p}_i, \mathbf{y}_i) + \alpha \cdot H(\mathbf{p}_i), \tag{4}$$

where $CE(\cdot,\cdot)$ is the cross entropy; $\mathbf{y}_i$ is label of sample i in one-hot representation; and α is the hyper-parameter for the entropy regularization. Tuning the value of α is simple. When the confidence underestimates the accuracy, we set $\alpha < 0$ and vice-versa. Our confidence calibration method is simple but works well in practice. Its evaluation is presented later in this paper.

B. Dynamic Utility Updates

We define the utility of executing a stage (of a task) as the expected increase in output confidence if the stage is executed (compared to the confidence in output before the stage is executed). In the previous section, we described how confidence is estimated once a stage is executed and its output obtained. It remains to describe how to estimate confidence in future stage outputs before these stages are executed. We predict confidence in results of future stages using regression models that relate computed confidence in results of previously executed stage(s) to predicted confidence in results of future stages. Specifically, we choose the Gaussian process regression model [16]. We made this choice for two reasons. First, the Gaussian process model is the state-of-the-art regression model. Second, Gaussian processes produce a Gaussian distribution as the output, from which we can easily compute the mean value and desired confidence intervals.

Using this approach, we gradually refine confidence during the execution of inference algorithms. At the beginning, predicted confidence in results is the same for all tasks, and is based on overall statistics computed from training data. However, as tasks compute results at intermediate stages, each task obtains an updated confidence in computed results and is thus able to update its estimate of confidence in subsequent stage results using the aforementioned regression model.

For a three-stage neural network, as shown in Figure 1, we train three Gaussian process regression models, $\hat{p}_i^{(2)} = \mathcal{GP}_{1\_2}(p_i^{(1)})$, $\hat{p}_i^{(3)} = \mathcal{GP}_{1\_3}(p_i^{(1)})$, and $\hat{p}_i^{(3)} = \mathcal{GP}_{2\_3}(p_i^{(2)})$, where $p_i^{(l)}$ denotes the classification confidence of sample i at neural network stage l. These regression models are learnt from the confidence curves of training data.

However, Gaussian process is notorious for its long inference time, which is unacceptable for a runtime predictor. Fortunately, the inputs of these gaussian models are bounded, *i.e.*, $p_i^{(l)} \in [0, 1]$. Therefore, we can approximate these complex Gaussian process regression models with simple piece-wise linear functions with two steps:

1) profiling the Gaussian process regression model with a set of input confidences, $\{0, 1/M, \cdots, 1\}$.
2) connecting these profiling points with a piece-wise linear function.

Thus, we can use these computationally efficient piece-wise linear functions at runtime for updating future stage confidence estimates dynamically.

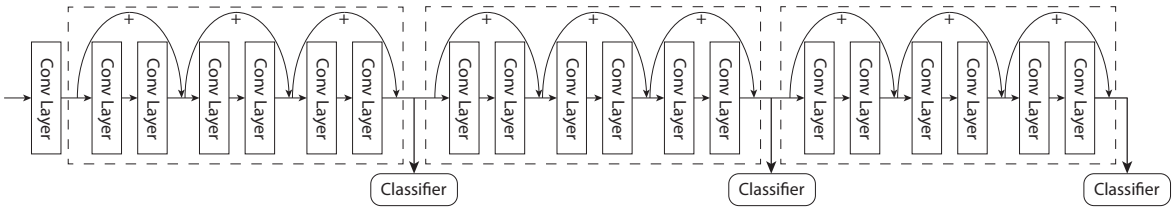


Fig. 3: The illustration of three-stage ResNet.

With predicted future confidence computed, we have all we need to do utility maximizing scheduling. As mentioned earlier, the scheduler picks the stage of a task whose execution will increase predicted confidence in results (i.e., utility) the most. Under certain conditions (submodular utility curves and equal stage execution times), the scheduler optimizes global utility of the service.

C. A Proof of Concept

The feasibility and efficacy of training [4], automatic labeling [8], model reduction [5], and profiling [9] functions have recently been studied and reported in the respective citations. Hence, below, we focus on the exploitation of trained networks; namely, the combination of quality estimation, inference, and run-time scheduling. We test these functions in the context of an image recognition service, implemented based on a state-of-the-art convolutional neural network (CNN) structure; namely, residual neural networks (ResNet). As shown in Figure 3, compared to traditional CNNs, ResNets add extra shortcut connections between convolutional layers. The whole ResNet is divided into three stages. Except for the bottom convolutional layer on the left side, each stage consists of six convolutional layers with three residual shortcut connections. At the end of each stage, a simple softmax classifier is appended, using the end-of-stage aggregated features for classification. The network was trained on the CIFAR-10 dataset with 50000 training images.

The scheduler spawns a pool of worker processes. These processes wait on input images to arrive. Each image represents a task submitted to the system. When an input image arrives, it is assigned to a process in the pool. The process runs the aforementioned deep neural network on the new input. The execution of the process features an explicit separation into stages. A stage might contain multiple layers. When finished, each stage will output a tuple in the form *(predicted value, confidence)*. *Predicted value* is the classification result from the current stage, specifying the most likely classification. *Confidence* describes the likelihood that this classification is correct. For example, a picture can be classified as a cat, dog, or cow, with probabilities 0.6, 0.3, and 0.1, respectively. The classification result is then ("cat", 0.6). The *confidence* in classification will then be sent to our user-level scheduler through a named pipe in linux. When a task is finished, the corresponding process is returned to the pool.

Note that, since our greedy algorithm tends to choose stages with the maximum incremental utility for future execution, tasks with lower initial classification confidence values tend to be selected for another execution stage. This has the side-effect of attaining better fairness as well.

To verify the effectiveness of this scheduling algorithm, we test the scheduler with several processes running the aforementioned residual neural network. Each process classifies images from the CIFAR-10 dataset, not included in the training set. The dataset contains images of 10 classes. Images arrive in a randomly shuffled order. The workstation that runs the scheduler and the classification processes has 8 Intel i7-4770 CPUs, with 32 GB memory. The evaluation is performed under Ubuntu 16.04 with kernel version 4.13. The residual neural network is implemented on TensorFlow 1.4.0.

Confidence Calibration & Dynamic Updates: In this experiment, we compare the following three confidence calibration methods:

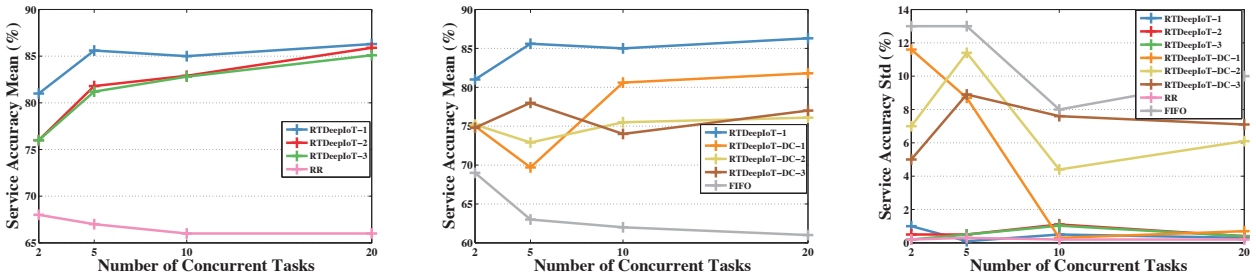
1) RTDeepIoT: It refers to the entropy-based confidence calibration method described in Equation (4).
2) RDeepSense: A state-of-the-art confidence calibration method with dropout operations [6].
3) Uncalibrated: The original confidence estimates without calibration.

The resulting ECE metric, defined in Equation (3), is shown in Table II. RTDeepIoT achieves the smallest ECE among all three stages, even compared to the state-of-the-art RDeepSense method. The evaluation results show that the proposed simple entropy-based confidence calibration method can provide a good estimation of classification accuracy, making it possible for the RTDeepIoT scheduler to optimize utility in a more informed fashion.

TABLE II: The ECE of confidence calibration methods with three-stage ResNet on CIFAR-10 dataset .

	Uncalibrated	RDeepSense	RTDeepIoT
Stage 1	0.134	0.058	**0.010**
Stage 2	0.146	0.046	**0.012**
Stage 3	0.123	0.054	**0.008**

Next, we evaluate the quality of prediction of confidence in results of future execution stages, based on the three regression models, $\hat{p}_i^{(2)} = \mathcal{GP}_{1\_2}(p_i^{(1)})$, $\hat{p}_i^{(3)} = \mathcal{GP}_{1\_3}(p_i^{(1)})$, and $\hat{p}_i^{(3)} = \mathcal{GP}_{2\_3}(p_i^{(2)})$. The evaluation results on Mean Absolute Error (MAE) and coefficient of determination (R^2) are shown in Table III. Overall, the method provides acceptable prediction



(a) The mean of service classification accuracy over RTDeepIoT-k and RR.

(b) The mean of service classification accuracy over RTDeepIoT-1, RTDeepIoT-DC-k, and FIFO.

(c) The standard deviation of service classification accuracy.

Fig. 4: The scalability test for scheduling algorithms with ResNet on CIFAR-10.

result. As the number of finished stages increases, the dynamic prediction improves. That's to say, $\mathcal{GP}_{2\_3}$ has the lowest error, since it is used after the first two stages have already been executed, thereby offering more accurate predictions in the results of stage three.

TABLE III: The Mean Absolute Error (MAE) and coefficient of determination (R^2) of dynamic confidence curve prediction for three-stage ResNet on CIFAR-10 dataset .

	$\mathcal{GP}_{1\_2}$	$\mathcal{GP}_{1\_3}$	$\mathcal{GP}_{2\_3}$
MAE	0.124	0.108	0.072
R^2	0.57	0.43	0.78

Runtime Scheduling: Next, we evaluate the effectiveness of the run-time scheduler that attempts to optimize utility by picking the stage that maximizes the increase in predicted confidence next. Specifically, we compare the following scheduling variants:

1) RTDeepIoT-k: this is our novel scheduler, where k the lookahead parameter mentioned in Section III.
2) RTDeepIoT-DC-k: this is a simplified variant of our scheduler. Instead of using dynamic confidence updates, it assumes that the confidence will continue to increase with the same slope. Therefore, it uses the confidence increase in the current stage as the predicted increase per each of the future stages.
3) RR: this is a stage-level round-robin scheduling algorithm. The scheduler will select a stage to run among all the deep learning services in a round-robin manner.
4) FIFO: this is a FIFO scheduling algorithm, where the scheduler runs the deep learning service on images in a first come first served manner, and runs all stages to the end.

The results of the experiment are shown in Figure 4 (where for readibility we break the baselines into two subsets and compare them in two different subfigures). Both show that our scheduling policy does better than simpler baselines, especially FIFO and RR scheduling. The standard deviation of classification accuracy is shown in Figure 4c. It reveals divergence between two types of algorithms. A lower deviation means better fairness. Our scheduling algorithm can balance the computation fairly, even with a very biased utility curve.

IV. Next: Collaborative Inferencing

The services described until now operate largely in a *per-device* fashion. Services such as training, compression, caching or profiling are offered to each individual IoT device's neural network pipelines. However, in many environments, IoT devices are not deployed individually, but rather as a *collection* of possibly heterogeneous nodes that together support an application. In this distributed environment, how can one support such collaborative inferencing?

A. Distributing the Inference Model

In the simplest case, the multistage nature of neural networks allows for an interesting possibility to share the load between clients and servers (besides caching, described earlier in the paper). Namely, in performing inference, it may be possible to execute some stages of the neural network on the client, leaving other stages to execute on the server. If the confidence in results obtained on the client is sufficiently high, no subsequent offloading to the server is needed. Otherwise, processing continues on the server. The approach raises questions regarding optimal partitioning of the model between the client and server. An ideal partitioning should maximally reduce client reliance on remote processing on the server, while observing client-side resource constraints as well as communication bandwidth constraints between the client and server.

An extension of this collaboration model is one where multiple distributed sensors (the clients) contribute data to be collectively used as input to the inference process. In one realization, clients would send their raw data to the server. The server would execute the entire neural network model on received data from all clients in order to compute inference results. In many cases, however, it may be more efficient for clients to execute some part of the inference network locally on their own data then send intermediate results to the server to continue model execution remotely. In the latter case, how should the inference model be partitioned among nodes in the distributed system? Optimal partitioning can take into account

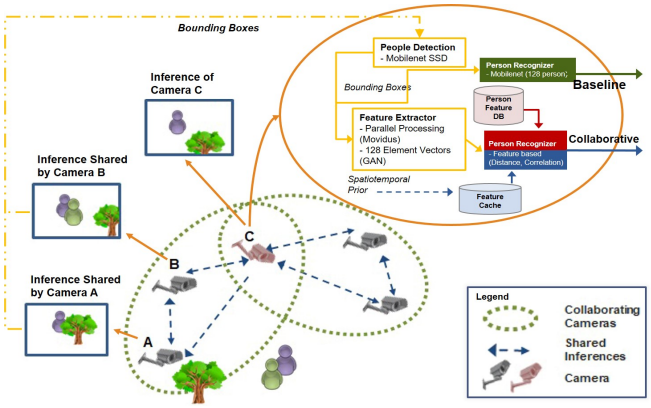


Fig. 5: Collaborative IoT (Camera) environment & Deep Inferencing Pipelines.

resources available on individual nodes, communication bandwidth among them, as well as any end-to-end requirements such as maximum allowable latency. Viewing neural network models as the intermediate code representation for a virtual machine implies potential for great flexibility in how execution is partitioned in the distributed system. Adaptive algorithms are needed to maximally exploit this flexibility (e.g., in mobile or dynamic environments) where connectivity, power, and other local resources may change over time.

B. Orchestrating Collaboration

A more interesting form of cooperative processing is one where the distributed devices cooperate to mutually enhance each other's performance. For example, two cameras may realize that they are looking at the same target (e.g., because of the way they are positioned, and because of the location of the target in their respective fields of view). Hence, rather than performing target classification twice in two independent tasks, each running on inputs from one of the cameras, it might be possible to join the tasks for better accuracy. How and when should one perform such a join to best enhance classification results based on the collective data of both cameras? Note that, individually, the two cameras might not have enough information to conclude that they are observing the same target (e.g., they might not know that they have overlapping fields of view). However, the server, observing classification outputs of the two cameras over time, may conclude that their fields of view are indeed overlapping. This knowledge can thereafter be used to determine if their outputs should be processed jointly to improve accuracy of classification. The same wisdom may apply to sensors of different modalities, such as microphones and vibration sensors. In short, an edge server offering intelligence as a service for a number of IoT devices may serve the additional function of discovering correlations among their data (e.g., inferred from correlations in produced labels) that can thereafter be used to reconfigure, and possibly re-train, the neural network model to better exploit the data from these correlated sources.

Consider, for example, a set of surveillance video cameras, deployed across a smart university campus (as illustrated in Figure 5) to support applications such as people *counting* (estimating the aggregated occupancy in different parts of the campus) or people *tracking* (capturing the movement trajectory of a specific individual throughout the campus). Conventionally, we can envisage that each camera operates as an *isolated* IoT device, applying state-of-the-art DNN-based techniques, such as *MobileNet* Single-Shot Detectors (SSD) [17], to perform object (people) detection, followed by object (people) identification, on each frame. Such an approach, however, has two limitations: (a) *poor processing efficiency*: executing 2 independent DNNs even on a specialized edge node (e.g., Intel's Movidius^{TM}neuromorphic co-processor) consumes ≈ 550 msecs/frame, implying a processing throughput $\leq$ 2fps; (b) *lower accuracy*: individual cameras may often be affected by specific context-based artifacts (e.g., occlusions, poor lighting) that impair the object detection process.

TABLE IV: Collaborative Deep IoT Inferencing

Approach	Detection Accuracy	Recognition Latency
Individual	68%	550 msec
Collaborative	75.5%	25 msec

To overcome these limitations, it is possible to explore the notion of *collaborative inferencing*, where the inferencing pipelines of different IoT devices exchange *state information* in near real time and subsequently adapt their individual execution logic. As a specific illustrative example, consider Figure 5, where each camera has a field-of-view (*FoV*) with varying degrees of overlap with neighboring cameras–e.g., cameras B and C both observe two individuals and a tree (from different perspectives) concurrently. In this scenario, the cameras may collaborate to improve their overall operational efficiency and accuracy. For example, one camera that detects individual bounding boxes (individuals) in its FoV may share those bounding box coordinates with its neighboring cameras. The other peer cameras can then supplement their own DNN-based inferences with these additional object coordinates (suitably remapped to a common coordinate space) to improve both their detection accuracy (for people counting) and reduce their processing latency (for individual tracking).

The collaborative paradigm described above was evaluated with the PETS dataset [18], consisting of 8 outdoor cameras. Table IV summarizes the performance differences between the baseline (non-collaborative) vs. the collaborative deep inferencing approach. We see that such collaboration is indeed beneficial: it increases the people counting accuracy by $\geq$8%, and achieves a 20-fold reduction in the average per-frame processing latency.

C. Services for Collaborative Inferencing

To realize the benefits of such collaborative deep inferencing, we believe that it will be important to augment *Eugene* to provide several new forms of functionality. These include:

- *Collaboration Brokering:* The collaborative video monitoring example provided earlier implicitly assumes that the cameras are aware of each other's identity & the

extent of FoV overlap. Note that such overlap need not be concurrent: one can envisage future scenarios where the camera views are temporally correlated with a variable lag-e.g., two corridors at two ends of a campus building corridor are likely to observe the same individuals 20 seconds apart. To easily support such dense IoT deployments, it is necessary to discover such correlations, and establish the identity of collaborators, in a more autonomous fashion. This is where *Eugene* can step in: by operating on the metadata & higher-level inferences from individual nodes, *Eugene* can discover and establish the relevant collaboration parameters–e.g., instructing cameras A & B to apply the collaborative tracking mechanism discussed above, but with a time lag of 20 seconds. Developing suitable mechanisms that uncover such useful spatiotemporal correlations among IoT devices, while satisfying the requirements of low communication overheads and privacy, is an open challenge.

- *Resilient Collaboration:* Collaborative deep inferencing, however, introduces a new form of failure: their operation is vulnerable to incorrect or malicious behavior by individual IoT nodes. For example, false or noisy bounding box estimates by one camera can reduce the people detection accuracy of other peer cameras by over 20%. To promote practical use of such collaboration paradigms, *Eugene* must also provide *resiliency services* that provide protection against such adversarial behavior. One can imagine a future where *Eugene* continuously monitors the output inference streams, and the internal parameters of relevant deep pipelines, of individual IoT devices to first (a) proactively uncover faulty operational situations and subsequently (b) provide suitable pipeline modifications to compensate for such faults.

V. Conclusions and Future Work

This paper envisioned (and described the current status of) a novel service model, called "intelligence as a service", to empower future IoT applications, where simple devices with sensing capabilities offload their machine intelligence needs to the cloud or to an edge server, possibly caching reduced models. We focused on deep learning as the state of the art enabler of machine intelligence. Several service components were presented together with related challenges at both training and inference time. As an example of system support for this service, a scheduler was described that optimizes service utility. Preliminary evaluation results were reported, as well as opportunities for further work; most importantly, collaborative inferencing.

The work opens many related research opportunities. For example, the paper did not explicitly discuss service models and APIs. Where will training data and labels come from? One service model would be to define data pools (e.g., the "Downtown Mall's Security Cameras Pool"). Only devices authorized to contribute to the pool can add data and/or labels to it for purposes of neural network model training.

A question that arises when multiple devices collaborate on the same data is how to handle rogue devices (or insider attacks) that gain access to the data for the purpose of polluting the pool with adversarial inputs (e.g., bad samples or wrong labels)? Some form of anomaly detection may be needed in order to identify input samples that differ from the rest. For example, if samples arriving from one of the devices are often misclassified based on models computed from other devices' data, then one may suspect rogue behavior. Efficient solutions are needed to implement such tests given that the number and identity of rogue devices are unknown. How to handle malicious devices that mix bad inputs with some amounts of good data to avoid suspicion?

The service, as described in this paper, treats all client devices alike and aims to offer fairness (e.g., imposes the same maximum allowable latency constraint on all tasks). In reality, different applications will have different demands and constraints. For example, an interactive voice chatbot might have significantly tighter latency constraints than an intrusion detection camera. A few seconds of delay in detecting suspicious behavior is tolerable, but a similar delay before each response in an interactive conversation might be very distracting. The scheduler described in this paper needs to be modified to support multiple service classes and account for different execution cost and constraints. An appropriate pricing structure may be needed that is informed of the true resource cost imposed by clients of each class on the service.

We hope the work reported in this paper will help produce early prototypes of machine intelligence services for IoT systems, and contribute to the realization of a new smart edge, where each device appears endowed with unlimited knowledge and intelligent behavior. Indeed, understanding the true potential, capabilities, and limitations of intelligence as a service may be the first step towards revolutionizing our interactions with physical surroundings in the near future. The authors hope that this paper makes a step towards such an understanding, if only by formulating some of the questions whose answers are to be understood.

Acknowledgements

This material is supported partially by the National Research Foundation, Prime Minister's Office, Singapore under its International Research Centers in Singapore Funding Initiative. The research reported in this paper was also sponsored in part by NSF under grants CNS 16-18627 and CNS 13-20209 and in part by the US Army Research Laboratory under Cooperative Agreements W911NF-09-2-0053 and W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory, NSF, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, S. Lu, and T. Abdelzaher, "Deep learning for the internet of things," *Computer*, vol. 51, no. 5, pp. 32–41, May 2018. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MC.2018.2381131

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[4] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "Deepsense: a unified deep learning framework for time-series mobile sensing data processing," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017.

[5] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017.

[6] S. Yao, Y. Zhao, H. Shao, A. Zhang, C. Zhang, S. Li, and T. Abdelzaher, "Rdeepsense: Reliable deep mobile computing models with uncertainty estimations," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, p. 173, 2018.

[7] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 283–294.

[8] S. Yao, Y. Zhao, H. Shao, C. Zhang, A. Zhang, S. Hu, D. Liu, S. Liu, L. Su, and T. Abdelzaher, "Sensegan: Enabling deep learning for internet of things with a semi-supervised framework," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, pp. 144:1–144:21, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3264954

[9] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '18. New York, NY, USA: ACM, 2018, pp. 278–291. [Online]. Available: http://doi.acm.org/10.1145/3274783.3274840

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[11] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," *arXiv preprint arXiv:1706.04599*, 2017.

[12] M. H. DeGroot and S. E. Fienberg, "The comparison and evaluation of forecasters," *The statistician*, pp. 12–22, 1983.

[13] M. P. Naeini, G. F. Cooper, and M. Hauskrecht, "Obtaining well calibrated probabilities using bayesian binning." in *AAAI*, 2015, pp. 2901–2907.

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

[15] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *arXiv preprint arXiv:1612.01474*, 2016.

[16] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced lectures on machine learning*. Springer, 2004, pp. 63–71.

[17] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *ECCV*, 2016.

[18] J. Ferryman and A. Shahrokni, "Pets2009: Dataset and challenge," in *2009 Twelfth IEEE International Workshop on Performance Evaluation of Tracking and Surveillance*. IEEE, 2009.