Guardians of the Deep Fog: Failure-Resilient DNN Inference from Edge to Cloud

Ashkan Yousefpour*† UT Dallas & UC Berkeley
Siddartha Devic* UT Dallas
Brian Q. Nguyen* UT Dallas
Aboudy Kreidieh UC Berkeley
Alan Liao UT Dallas
Alexandre M. Bayen UC Berkeley
Jason P. Jue UT Dallas

ABSTRACT

Partitioning and distributing *deep neural networks* (DNNs) over physical nodes such as edge, fog, or cloud nodes, could enhance sensor fusion, and reduce bandwidth and inference latency. However, when a DNN is distributed over physical nodes, failure of the physical nodes causes the failure of the DNN units that are placed on these nodes. The performance of the inference task will be unpredictable, and most likely, poor, if the distributed DNN is not specifically designed and properly trained for failures. Motivated by this, we introduce *deepFogGuard*, a DNN architecture augmentation scheme for making the distributed DNN inference task failure-resilient. To articulate *deepFogGuard*, we introduce the elements and a model for the resiliency of distributed DNN inference. Inspired by the concept of residual connections in DNNs, we introduce skip hyperconnections in distributed DNNs, which are the basis of *deepFogGuard*'s design to provide resiliency. Next, our extensive experiments using two existing datasets for the sensing and vision applications confirm the ability of *deepFogGuard* to provide resiliency for distributed DNNs in edge-cloud networks.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Reliability**; *Distributed architectures.*

KEYWORDS

Fog Computing, Edge Computing, Distributed Neural Networks, Resiliency, Reliability, Robust, Distributed DNN Inference

ACM Reference Format:
Ashkan Yousefpour, Siddartha Devic, Brian Q. Nguyen, Aboudy Kreidieh, Alan Liao, Alexandre M. Bayen, and Jason P. Jue. 2019. *Guardians of the Deep Fog*: Failure-Resilient DNN Inference from Edge to Cloud. In *First International Workshop on Challenges in Artificial Intelligence and Machine Learning (AIChallengeIoT'19), November 10–13, 2019, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3363347.3363366

*These authors contributed equally to the paper.
†ashkan.y@berkeley.edu

AIChallengeIoT'19, November 10–13, 2019, New York, NY, USA

ACM ISBN 978-1-4503-7013-4/19/11…$15.00
https://doi.org/10.1145/3363347.3363366

1 INTRODUCTION

With the proliferation of the *Internet of Things* (IoT) applications, increasing numbers of smart IoT devices are being deployed and integrated into our daily routines. Smart homes, smart cities, wearables, self-driving vehicles, AR and VR, context sensing and crowdsensing, and smart retail are examples of adaptations of IoT devices into human spaces [33, 34]. To intelligently analyze and act on the data that IoT devices generate, *machine learning* (ML) techniques are seen to be promising. This is primarily because IoT devices are often directly connected to data sources, such as cameras, microphones, gyroscopes, or sensors that capture a large quantity of input data that could feed the ML models [2, 29].

Due to their accuracy and powerful expressiveness, deep learning methods, among other ML techniques, have been a successful choice for IoT applications in a broad spectrum of domains such as computer vision, speech recognition, medical diagnosis, and natural language processing [17, 21]. Deep learning techniques make use of *deep neural networks* (DNNs). In certain DNN-empowered IoT applications, the *inference* task runs for a prolonged period of time. Examples of such IoT applications are image-based defect detection in a factory, automatic recognition of parts during product assembly, or anomaly behavior detection in a crowd based on DNNs [6]. Nevertheless, a challenge with DNN-empowered IoT applications is determining where the DNN model should be placed for the inference task.

The immediate option may be deploying DNNs directly onto the IoT devices; however, this is often infeasible, as many IoT devices are resource-constrained and cannot efficiently support the computational requirements of DNNs. For instance, according to Liu *et al.* [18], the GoogleNet model for image classification is larger than 20 MB and requires about 1.5 billion multiply-add operations per inference per image.

Another possibility is to place the DNN in the cloud and send the IoT data to the cloud, since the cloud servers are equipped with powerful hardware such as TPUs and GPUs. Nevertheless, when a DNN is deployed in the cloud, the data has to be continuously transmitted from IoT devices to the cloud in WAN environments during inference, which results in the heavy consumption of network resources, high latency, and privacy concerns [18, 29].

Another option for DNN placement is to distribute the DNN over physical nodes along an edge-fog-cloud hierarchy [9, 14, 18, 20, 28, 29]. The idea of thie current approach is to distribute the DNN onto *edge nodes*, *fog nodes*, and *cloud nodes* so that inference from IoT data is processed along the route, on different physical nodes from the edge to cloud.

A natural question that arises with this approach is whether the resulting distributed DNN inference along edge-fog-cloud is resilient in the presence of physical nodes failures. Specifically, the question is: *what happens to an ongoing inference task of a distributed DNN when its physical nodes fail, and how can we make distributed DNN inference resilient to physical node failures?* This question is the topic of our study.

When a DNN is distributed over physical nodes, failure of a physical node causes the failure of the DNN units that are placed on the node. Failure of physical nodes could be due to power outages, cable cuts, natural disasters, or hardware/software failures. The effect of such failures on distributed DNN inference may be heavily dependent on the time to recover from the failures.

While the physical nodes are being recovered, the performance of the distributed DNN inference is unpredictable, and most likely, poor, if the distributed DNN is not specifically designed and properly trained for failure resiliency. This is especially important for critical applications that cannot tolerate unpredictable and poor performance, even for a short time.

In this article, we study the failure resiliency of distributed DNN inference over the edge, fog, and cloud nodes, where the failure of a physical node results in the failure of the DNN units that are placed on the node. Our main contributions in this article are

(1) *We introduce deepFogGuard, a DNN architecture augmentation scheme for making the distributed DNN inference failure-resilient*: In order to provide context for *deepFogGuard*, we introduce the elements of distributed DNNs and *deepFogGuard*. Inspired by the concept of residual connections in DNNs [13], we introduce **skip hyperconnections** in distributed DNNs, which are the basis of *deepFogGuard*'s design to provide resiliency. Residual connections skip one or more DNN layers, whereas skip hyperconnections skip one or more physical nodes.
(2) *We conduct extensive experiments using two existing data sets for sensing and vision applications*: We construct a model for measuring the resiliency of inference in distributed DNNs. Finally, we confirm the ability of *deepFogGuard* to provide resiliency for distributed DNNs in edge-cloud networks.

In *deepFogGuard*, upon failure of a physical node, the information flow can still be routed through the distributed DNN, thanks to the skip hyperconnections. Hence, we call the skip hyperconnections the *Guardians of the Deep Fog*, since they act as the guard of information flow in distributed DNNs over edge-fog-cloud hierarchy.

2 DEFINITIONS

In this section we introduce the definitions required to articulate *deepFogGuard* and provide the necessary context.

Partitioning and distributing DNNs. A deep neural network G can be split according to a partition map u and can be distributed over a set of physical nodes V. We denote the partition operation by $\oslash$ and the resulting split DNN by $G \oslash^{u} V$. The present study does not address the problem of optimal partitioning of the DNNs (i.e. finding an optimal partition map u), as it is not the primary focus of this article. The optimal DNN partitioning is non-trivial and depends on many factors including available network bandwidth, type of DNN layers (convolutional vs. fully-connected), and DNN graph topology [9, 11, 14, 15, 17, 32, 35]. Instead, this article studies the resiliency of *previously-partitioned* distributed DNN models during inference.

Previously partitioned DNN. If u_* denotes the desired partition map for a certain use case (with regards to constraints such as delay, bandwidth, or energy), G_V denotes the resulting partitioned DNN according to u_*. Hence, $G_V = G \oslash^{u_*} V$.

Physical Nodes vs. DNN Units: To distinguish between physical nodes in the network and DNN nodes (units or neurons), we clarify by using *units* when referring to DNN neurons. Additionally, we only use the term *physical node*; however, the concepts in this study are also applicable to *virtual nodes*, such as VMs or containers.

Types of Physical Nodes: IoT devices (e.g. sensors, cameras, or mobile phones) are usually the main sources of data, whereas cloud servers are central hubs for processing and storage. Cloud servers are normally part of large data centers. On the other hand, fog nodes could host services packaged in the form of VMs, containers, or unikernels, and can be routers, switches, dedicated servers for fog computing (e.g. *cloudlets*), set-top boxes, access points, or firewalls. Similarly, edge nodes are devices attached to the connected things, such as WiFi routers, switches, and base stations [34].

Figure 1 shows the process of splitting a DNN with four fully-connected layers and distributing it across two physical nodes v_1 and v_2. The layers l_1 and l_2 are stored on node v_2 and the layers l_3 and l_4 are stored on node v_1. $\mathbf{W}^{(3)}$ is matrix of weights at layer l_3. Note that a special layer $l_1^{\#}$ (called *expansion layer*) is included below layer l_3 in node v_1. This layer is for the reception and the expansion of the vector of data from the other physical nodes (to be discussed). Since the distributed DNN resides on different physical nodes, during inference the vector of output values from one physical node must be transferred (e.g. through a TCP socket) to another physical node. We call the transfer link (pipe) between two physical nodes a *hyperconnection*.

Hyperconnections: Unlike a typical neural network that connects two units and transfers a scalar, a *hyperconnection* connects two physical nodes over which the layers of a DNN is distributed and transfers a vector of scalars.

Simple vs. Skip Hyperconnections: Hyperconnections are one of two kinds: *simple* or *skip*. A hyperconnection is called *simple* when it connects a physical node to the physical node that has the next DNN layer (i.e. "parent" node in the hierarchy), and is called *skip* when it skips one or more physical nodes in the hierarchy and connects a physical node to an "ancestor" node. In Fig. 1, the simple hyperconnection between v_1 and v_2 connects the output values of layer l_2 to the input of expansion layer $l_1^{\#}$. The skip hyperconnection connects the output of a "descendant" physical node v_j (not shown) to the input of expansion layer $l_1^{\#}$.

The concept of skip hyperconnection is similar to that of residual connections in DNNs [13]. Residual connections are a special case of highway connections [13, 27] and are those connections skipping one or more DNN layers. Similarly, skip hyperconnections skip one or more physical nodes in a distributed DNN. *deepFogGuard* makes use of skip hyperconnections for added resiliency, so that upon failure of a physical node, the information flow can still be routed to the cloud for inference. In Section 3 we explain that adding skip hyperconnections improves resiliency of the distributed DNN.

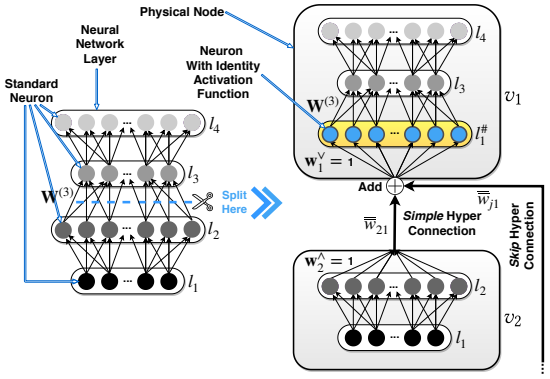

Figure 1: Partitioning a DNN and distributing it across physical nodes v_1 and v_2. The DNN is fully connected (not all weights are shown).

Remark: Note that the weights that feed the unit output values of physical node v_2 to the hyperconnection (vector $\mathbf{w}_2^{\wedge}$) and the weights that expand the output of the *Add* operation to the expansion layer of physical node v_1 (vector $\mathbf{w}_1^{\vee}$) are all set to 1. The value of these vectors can be chosen arbitrarily because distributed DNNs learn to adjust and compensate their weights during training (the training process will be discussed soon). For simplicity but without loss of generality, we assume the value of **1** for these vectors.

Hyperconnection Weights: Similar to connection weights in neural networks, hyperconnections also have a *weight* vector; the elements of the vector that passes through the hyperconnection are multiplied by this weight. Let vector $\overline{\overline{\mathbf{w}}}_{ij}$ denote the hyperconnection weight that connects physical node v_i to physical node v_j. In this study, $\forall i, j\ \overline{\overline{\mathbf{w}}}_{ij} = \mathbf{1}$, that is the weight of all hyperconnections is chosen to be the vector **1** (a vector with all elements equal to 1).

Definition 1. *Adding hyperconnections' inputs.* The *Add* $\bigoplus$ operation is an element-wise vector addition that adds the elements of two or more hyperconnections. When H_i denotes the set of indices of all physical node that have a hyperconnection to the physical node v_i, the Add operation at node v_i computes the vector $\mathbf{x}^{(l_i^{\#})}$, the input vector to the expansion layer of node v_i, by adding the data going through the incoming hyperconnections as

$$\mathbf{x}^{(l_i^{\#})} = \sum_{j \in H_i} \overline{\overline{\mathbf{w}}}_{ji} \mathbf{x}_{ji}, \tag{1}$$

where $\mathbf{x}_{ji}$ is the vector passing through the hyperconnection connecting the physical node v_j to the physical node v_i. Since, in this article, $\overline{\overline{\mathbf{w}}}_{ij} = \mathbf{1}$, we will have $\mathbf{x}^{(l_i^{\#})} = \sum_{j \in H_i} \mathbf{x}_{ji}$.

We represent the vector output of a failed physical node i.e. *null vector*, by the symbol Φ. Formally, when a null vector is added to a non-null vector, the null vector is ignored. That is, $\mathbf{x}_{ij} \bigoplus \Phi = \mathbf{x}_{ij}$. In the case where all source physical nodes of the incoming hyperconnections to a physical node fail, we will have $\Phi \bigoplus \ldots \bigoplus \Phi = \Phi$, which means the input of the physical node will be the null vector. Since in this case, applying operations on the null vector is meaningless, the physical node outputs the null vector Φ. If the null vector is propagated all the way through the last layer of DNN, which means the information flow did not make it to the last layer, random guessing is performed.

Hyperconnection Dimensions: The Add operation requires that the dimensions of the operands be the same. However, the dimensions of the hyperconnections are not always the same, as they connect DNN layers of different depth. When it is necessary to change the dimensions, we perform zero-padding to the vector(s) with smaller dimension. The vector with the largest dimension also decides the dimension (number of units) of the *expansion layer*.

Expansion Layer: A special layer called the *expansion layer* is added to a physical node for the reception of the input vector from the Add operation ($\mathbf{x}^{(l_i^{\#})}$) and its expansion to the units of the first layer at the physical node. The output (vector) of the Add operation must be connected to the corresponding units in the expansion layer. Hence, each unit in the expansion layer has one input, the corresponding value in the output vector. The units in the expansion layer all have identity function as their activation function, so that they do not change the incoming data. The number of the units in the expansion layer should be equal to the number of elements in the output vector of the Add operation, which is the maximum among the dimensions of the added vectors going through hyperconnections.

Training Process: Once the elements of the distributed DNN (hyperconnections and expansion layers) are added, the distributed DNN must be trained with the new elements. The training process need not to be distributed, that is, when the DNN is physically distributed; the training process can be executed in a "simulated" environment, where the DNN is distributed in a simulation.

3 DEEPFOGGUARD DESIGN

Distributed DNNs cannot be considered intrinsically failure-resilient without a proper design. Conventionally, we refer to the distributed DNNs that are not trained for failure resiliency as *Vanilla*.

deepFogGuard's goal is to increase the resiliency of the distributed DNN. We tend towards *passive resiliency*, the ability to function in the presence of failure without any re-training or reacting, but by exploiting the intrinsic resiliency [30]. To accomplish this, we consider and experiment with a method that augments the training process with built-in resiliency: adding skip hyperconnections to the architecture of a distributed DNN. Skip hyperconnections inherently increase the resiliency of the underlying neural architecture.

Resiliency via Skip Hyperconnections: The concept of skip hyperconnections is similar to residual connections. DNNs with residual connections are easier to optimize and have been shown to implicitly deal with the exploding gradient problem, ultimately providing better performance than standard DNNs [13]. Inspired by residual connections in DNNs, in *deepFogGuard* we add skip hyperconnection between physical nodes to provide additional pathways for the flow of information through the model, even in the presence of partial failures. This is to thwart the *no-information-flow* situation, in which the information does not make it to the last layer of the DNN, and random guessing has to be performed.

Figure 2 depicts the architecture of our experiments, discussed in Section 4. The dashed arrows represent skip hyperconnections between physical nodes. (The expansion layers are not shown in Fig. 2, since the layers have the same dimension in each experiment.) In

the architecture on the right in Fig. 2, all seven skip hyperconnections that skip one physical node are present. Similarly, all the three skip hyperconnections that skip one physical node are present in the distributed DNN on the left in Fig. 2. Generally, it is expected that more skip hyperconnections improve resiliency, especially in more extreme failure scenarios. Nevertheless, we found that the determining the number of skip hyperconnections is a non-trivial task and depends on the learning task, reliability setting, and original DNN architecture. Determining the right skip hyperconnections could be done during training. Resource heterogeneity across edge, fog, and cloud nodes could also be a deciding factor for setting up the skip hyperconnections.

Implementation Notes: (1) Skip hyperconnection can be implemented via TCP connections. (2) When implementing the Add operation, one has to ensure that the failure of a physical node does not result in an interruption, e.g. when a `TCP socket exception` is thrown. (3) When a physical node fails, another physical node should be able to "sense" the failure. Physical nodes are responsible for checking the hyperconnections' respective source nodes from which they are fed. This can be done through a simple *keep-alive* mechanism. (4) Inference should be implemented in a synchronized fashion, that is, if a physical node fails and its output is not present, the null vector Φ should be used as the data.

4 EXPERIMENTS

We need to construct a metric for measuring the resiliency of distributed DNNs. In the following subsection, we define a metric based on average accuracy to model resiliency of distributed DNNs.

4.1 Modeling Resiliency for Distributed DNN

We consider a set of n physical nodes $V = \{v_1, v_2, \ldots, v_n\}$ over which a DNN is distributed according to the partition map u.

Definition 2. *Reliability setting* of the physical nodes in V, is an n-tuple $R_V = (r_1, r_2, \ldots, r_n) \in [0,1]^n$, where each element r_i, referred to as *reliability probability*, is the probability that the physical node $v_i \in V$ survives at inference time. $r_i = (1 - p_i)$, where $p_i \in [0,1]$ is the probability that the physical node $v_i \in V$ fails during inference.

For the sake of easier terminology and notation, we utilize the convention in network reliability engineering, and use *reliability* over *probability of failure*. In order to model the reliability of a given distributed DNN, we need to model the following: (I) the physical nodes failing simultaneously during the inference time; and (II) the probability of a given simultaneous failure. To model (I), we introduce *node failure combination*, which is the combination of the physical nodes that fail simultaneously during the inference time.

Definition 3. A *node failure combination* of the physical nodes in V, is an n-tuple $B_V = (b_1, b_2, \ldots, b_n) \in \{0,1\}^n$, in which each element b_i is a binary value indicating whether the physical node $v_i \in V$ has failed (0) or not (1).

We now have a model for (I), through B_V. Since, at inference time, a physical node $v_i \in V$ can either fail or survive and since $|V| = n$, there are 2^n possible node failure combinations (multiple physical nodes can fail at the same time).

To model (II), we introduce $p(B_V|R_V)$, the probability of occurrence of a certain node failure combination B_V during inference time, given a reliability setting R_V over physical nodes V. This probability can be calculated as:

$$p(B_V|R_V) = \prod_{i=1}^{|V|} \big[b_i r_i + (1 - b_i)(1 - r_i)\big]. \tag{2}$$

As a numerical example, assume that a network has four physical nodes and reliability setting is $R_V = (0.98, 0.98, 0.95, 0.94)$. The probability of the node failure combination where node 4 fails ($B_V = (1,1,1,0)$) is $p(B_V|R_V) = 0.98 \times 0.98 \times 0.95 \times 0.06$.

Now we have a model for reliability of distributed DNNs. Next, we need to define a new notation $\varnothing$, necessary to model the resiliency of a distributed DNN:

Definition 4. *Operator* $\varnothing$. Given a node failure combination B_V, the notation $G_V \varnothing B_V$ represents the distributed DNN G_V (over physical nodes V), in which those units that reside on B_V's failing physical nodes are failed.

Average Accuracy as Resiliency Measure: For modeling the resiliency of distributed DNNs, performance indicators such as accuracy, precision, or recall may be used. In this article, we use *average accuracy* as the measure of resiliency.

The average accuracy of a distributed DNN G_V over physical nodes V against reliability setting R_V during inference is

$$\overline{\mathbb{A}}(G_V, R_V) = \sum_{B_V} p(B_V|R_V) \times \mathcal{A}(G_V \varnothing B_V), \tag{3}$$

where $\mathcal{A}(G_V \varnothing B_V)$ is the *accuracy* of $G_V \varnothing B_V$ during inference, and $p(B_V|R_V)$ is the probability of node failure combination B_V given a reliability setting R_V. Equation (3) calculates the weighted average

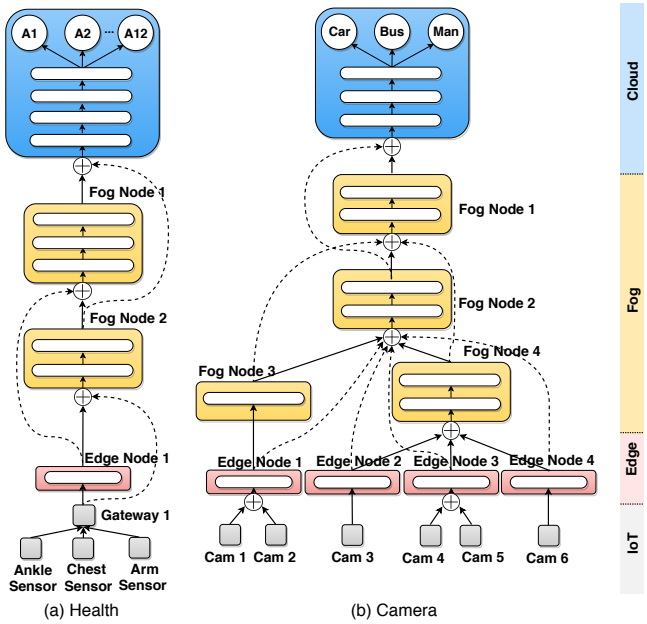

Figure 2: Distributed DNN architecture in *deepFogGuard* for: (a) health activity classification, (b) multi-camera object classification experiments. The expansion layers are not shown. (Physical node numbers are unique within each experiment)

Table 1: Reliability settings for all experiments.

	Reliability Setting R_V	
Experiment	Health (order: $[f_1, f_2, e_1]$)	Camera (order: $[f_1, f_2, f_3, f_4, e_1, e_2, e_3, e_4]$)
Surviv. Setting		
Normal	[99%, 98%, 96%]	[99.5%, 99%, 98%, 97%, 95%, 95%, 95%, 95%]
Poor	[98%, 96%, 92%]	[99%, 98%, 94%, 93%, 90%, 90%, 87%, 87%]
Hazardous	[90%, 85%, 80%]	[90%, 90%, 80%, 80%, 70%, 60%, 70%, 66%]

of accuracy over all possible node failure combinations (weighed by $p(B_V|R_V)$). Now we have a model for measuring resiliency of distributed DNNs. Next, we discus the datasets, the setup, and the results of our extensive experiment.

4.2 Datasets

Our extensive experiments are conducted using two existing datasets for the sensing and vision applications, explained as follows.

Health Activity Classification ("Health"): We utilize the mobile health activity sensor dataset (UCI *MHealth* [3]) as a benchmark for failure resiliency of a vertically distributed DNN (see Fig. 2a). The dataset is comprised of readings from various sensors placed on different body parts of patients. This dataset is an example of an IoT application for medical purposes. The dataset contains sensor acceleration data from three different sensors placed at the chest, left ankle, and right arm. Additionally, the left ankle and right arm sensor provide body orientation data, and the chest sensors provide ECG measurements. The dataset is labeled with the 12 activities a patient is performing at a given time, and the task is to classify the type of activity (e.g. if the patient is sitting or running). There are a total of 23 features, where each feature corresponds to a specific type of data collected from one of the three sensors across ten human test subjects. For this experiment, the activities that do not belong to one of the 12 classes are removed, resulting in a dataset of 343,185 data points. The health activity classification dataset is approximately uniformly distributed across each class after preprocessing, and hence we use a standard cross-entropy loss function for the classification.

Multi-Camera Object Classification ("Camera"): The multi-view object detection dataset [24] is used as a benchmark for failure resiliency in DNNs that are distributed both vertically and horizontally (see Fig. 2b). The dataset contains videos of a street from six different viewpoints, with object bounding boxes for frames captured from each camera. The bounding boxes are placed around three classes of objects: pedestrians, cars, and buses. Each camera is an IoT node, providing a viewpoint to the cloud for inference.

We crop the images to the bounding boxes to obtain (potentially) six different viewpoints for each object. We then center and resize each image to $32 \times 32 \times 3$ pixels, keeping the RGB channels. For views in which a particular object is obscured (and therefore does not exist in the object bounding box list), we generate an empty (black) image [29]. A single data instance is then a collection of six images of an object from the six cameras, and an associated label.

In total, the dataset contains around 1,400 data points. Since the distribution of the classes is skewed towards the "car" class, we utilize weighted (i.e. cost-sensitive) cross-entropy loss to incur a higher penalty for incorrectly predicting any given data point to be car [12]. Although the dataset contains very few images for the bus class, we keep it as one of the classes.

For both experiments, we separate each dataset into train, validation, and test with an 80/10/10 split. We use the validation set to select the best model among different *training epochs*. We run the model for many training epochs, select the model with highest validation accuracy, and report its accuracy on the test set, which the DNN has never seen.

4.3 Experiment Setup

We implemented our experiments on Google Cloud using TensorFlow and Keras. In order to assess the performance of our proposed method on different architectures of distributed DNNs, we propose separate model configurations for each experiment. For the health activity classification experiment (Fig. 2a), we consider a vertically distributed DNN that consists of ten hidden layers of width 250. The DNN layers are partitioned into four physical nodes (an edge node, two fog nodes, and a cloud node). The edge node contains one layer, the first fog node two, the second fog node three, and the cloud node four hidden layers. Conversely, for the multi-camera object classification experiment (Fig. 2b) we consider a DNN consisting of 14 layers of width 32 that are distributed both vertically and horizontally. The hidden layers are partitioned into nine physical nodes (four edge nodes, four fog nodes, and a cloud node). Images from the six individual cameras are merged using element-wise addition. We designed this highly distributed DNN architecture for the multi-camera object classification experiment, since it represents a very general architecture, where the DNN is distributed vertically and horizontally and is also asymmetric.

In our experiments, we only include the skip hyperconnections that skip one physical node, since, in the edge-fog-cloud hierarchy, the number of physical nodes over which the DNN is distributed is not large. Moreover, we experimented with models having skip hyperconnections that skip more than one physical node, but did not observe any performance gain.

Each of the aforementioned models is trained via stochastic gradient descent using the *Adam* optimizer [16]. Batch sizes of 1024 and 64, and the learning rates of 0.001 and 0.1 are used for the health activity classification and multi-camera object classification experiments, respectively.

We propose three different reliability settings outlined in Table 1: the setting *Normal* indicates reasonable network survivabilities while the settings *Poor* and *Hazardous* represent reliability settings (only for the sake of our experiments) when the failures are very frequent in the network. We also have the reliability setting *No Failure*, in which all survivabilities are simply set to 1. The failure probabilities of fog nodes and edge nodes in both experiments are described in Table 1. Fog nodes are denoted with f_i and edge nodes with e_i. We assume that the cloud node is always available and does not fail. If IoT nodes fail, the distributed DNNs will have no input, and random guessing must be performed. Since we are studying the resiliency of distributed DNNs (and not their input), we did not consider the failure of IoT nodes.

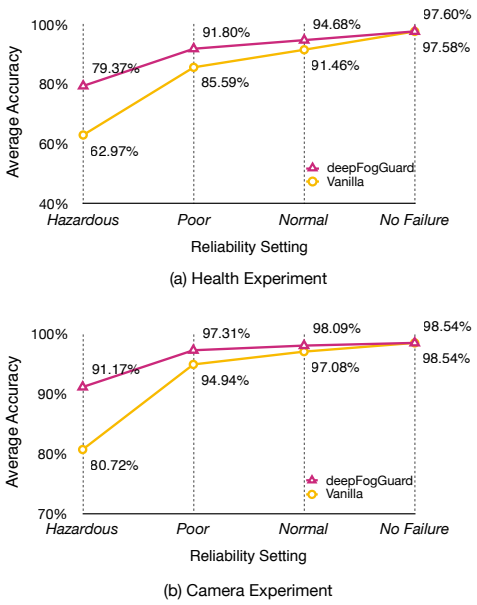

Figure 3: Average accuracy (%) vs. reliability setting

4.4 Results

Fig. 3 shows the average accuracy of *deepFogGuard* and Vanilla for each experiment under different reliability settings over 10 runs ($\overline{\mathbb{A}}(G_V, R_V)$). We can see that *deepFogGuard* is successful in increasing the resiliency of the distributed DNNs, drastically outperforming Vanilla. In the health activity classification experiment (Fig. 3a), in the *Hazardous* reliability setting, *deepFogGuard* increases average accuracy by almost 16%, relative to the baseline Vanilla. Similarly, in the *Poor* reliability setting, the average accuracy of *deepFogGuard* is around 6% higher than that of Vanilla. This difference in accuracy decreases when the reliability of the network is higher (e.g. in *Normal* or *No Failure*). Vanilla's poor performance under physical node failure is an indication of the inaccessibility of a path for information flow, hence the occurrence of random guessing for the classification task.

In the multi-camera object classification experiment, we observe the same trends as in the health activity classification experiment. Since the architecture of the DNNs is highly distributed (vertically and horizontally) in this experiment, even Vanilla can perform well (above 80%). This is because Vanilla's distributed DNN architecture still receives partial data in certain cases of failure. For instance, there are built-in redundancies among cameras, edge nodes, and fog nodes 3 and 4. This is not true for Vanilla in the health activity classification experiment, in which the distributed DNN is only vertically partitioned. Similar to the health activity classification experiment, we see the largest improvement in the *Hazardous* reliability setting, although *deepFogGuard* outperforms Vanilla across the board, in both experiments, under various reliability settings.

This concludes the discussion of our experiments. In the next section, we explain the state of the art in this direction, and we position our work's novelty in the literature. Finally, in Section 6, we discuss the limitations and opportunities to improve *deepFogGuard*.

5 RELATED WORK

The related work in this space can be categorized as follows.

a. Distributed training. Training of distributed DNNs has received significant attention from both academia and industry. Some examples include distributed training frameworks from Google [1], Facebook [22], Microsoft [7], and Uber [26]. Distributed training of DNNs across edge nodes is studied in [31] (non-resilient) and [5, 8] (resilient against adversaries). Nevertheless, inference in distributed DNNs is less explored. Recently, some IoT application scenarios have emerged that need ongoing and long inference tasks [9, 14, 18, 20, 28, 29]. In line with this direction, we study inference of distributed DNNs, but differently, we consider failure resiliency.

b. DNN Partitioning. DNN partitioning frameworks consider several factors to find the best partition map to split and distribute a DNN [9, 11, 14, 15, 17, 32, 35]. These frameworks can be used to provide input (partitioned distributed DNN) to *deepFogGuard*, from which resilient distributed DNN for inference is extracted.

c. DNN Fault Tolerance. In the DNN literature, a concept related to failure is *fault*, which is when units or weights become defective (i.e. stuck at a certain value, random bit flip, weight fault, or short circuit) [30]. Studies on fault tolerance of neural networks date back to the early 90s and are limited to mathematical models with simplistic assumptions (e.g. neural networks with one hidden layer, unit-only and weight-only faults, or sigmoid-only neural networks) [4, 19, 23]. However, none of these works consider the failure of physical nodes that potentially cause the failure of a large group of DNN units and weights.

d. DNN Failure Robustness and Resiliency. Some early works study the resiliency of *non-distributed* DNNs against *single or multiple* unit failures [25, 36]. More recently, the authors of [10] provide theoretical definitions and bounds for the failure of elements in non-distributed DNNs. Contrary to previous works, this article studies resiliency of *distributed* DNN inference in the presence of failure of a *large group of DNN units*.

6 CONCLUSION

We presented *deepFogGuard*, a method for failure resiliency of distributed DNN inference. We confirmed through experiments that skip hyperconnections increase the resiliency of distributed DNNs. *deepFogGuard* has a few limitations and opportunities for improvement, which we discuss below.

Limitations: While *deepFogGuard* improves resiliency by sending the data along redundant paths, it also inevitably consumes additional bandwidth when there are no failures. Moreover, keeping multiple TCP connections active and checking the status of other physical nodes consumes resources.

Future Work: This study opens many related research opportunities. Firstly, it is interesting to see how *deepFogGuard* can be extended to neural networks that have residual connections, or other types of neural networks, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Furthermore, regularization or methods that implicitly increase robustness, such as dropout, may improve the resiliency even more. Finally, one could consider changing the weights of the remaining hyperconnections after the failure of the physical nodes to account for the change in relative input scale of the physical node.

ACKNOWLEDGMENTS

We would like to thank professor Vibhav Gogate for the early discussions about resilient DNNs, and Ahmad Darki and professor Keith Winstein for their valuable ideas and comments.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Muhammad Ali, Ashiq Anjum, M Usman Yaseen, A Reza Zamani, Daniel Balouek-Thomert, Omer Rana, and Manish Parashar. 2018. Edge enhanced deep learning system for large-scale video stream analytics. In *IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 1–10.

[3] Oresti Banos, Claudia Villalonga, Rafael Garcia, Alejandro Saez, Miguel Damas, Juan A Holgado-Terriza, Sungyong Lee, Hector Pomares, and Ignacio Rojas. 2015. Design, implementation and validation of a novel open framework for agile development of mobile health applications. *Biomedical engineering online* 14, 2 (2015).

[4] George Ravuama Bolt. 1992. *Fault Tolerance in Artificial Neural Networks*. Ph.D. Dissertation. University of York.

[5] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. DRACO: Byzantine-resilient Distributed Training via Redundant Gradients. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. PMLR, 903–912. http://proceedings.mlr.press/v80/chen18l.html

[6] Yitao Chen, Kaiqi Zhao, Baoxin Li, and Ming Zhao. 2019. Exploring the Use of Synthetic Gradients for Distributed Deep Learning across Cloud and Edge Resources. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[7] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.

[8] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Hany Abdelmessih Guirguis, and SÃlbastien Louis Alexandre Rouault. 2019. AGGREGATHOR: Byzantine Machine Learning via Robust Gradient Aggregation. (2019). Conference on Systems and Machine Learning (SysML) 2019, Stanford, CA, USA.

[9] Swarnava Dey, Jayeeta Mondal, and Arijit Mukherjee. 2019. Offloaded Execution of Deep Learning Inference at Edge: Challenges and Insights. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 855–861.

[10] EM El Mhamdi, R Guerraoui, and S Rouault. 2017. On the robustness of a neural network. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 84–93.

[11] Tarek Elgamal, Atul Sandur, Phuong Nguyen, Klara Nahrstedt, and Gul Agha. 2018. DROPLET: Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 1–8.

[12] Haibo He and Edwardo A Garcia. 2008. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering* 9 (2008), 1263–1284.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[14] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1423–1431.

[15] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 615–629.

[16] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[17] En Li, Zhi Zhou, and Xu Chen. 2018. Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications (MECOMM'18)*. ACM, 31–36.

[18] Peng Liu, Bozhao Qi, and Suman Banerjee. 2018. EdgeEye: An Edge Service Framework for Real-time Intelligent Video Analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*. ACM, 1–6.

[19] Kishan Mehrotra, Chilukuri K Mohan, Sanjay Ranka, and Ching-tai Chiu. 1994. *Fault tolerance of neural networks*. Technical Report. Syracuse University. Tech. Rep. RL-TR-94-93. Syracuse University.

[20] Ahsan Morshed, Prem Prakash Jayaraman, Timos Sellis, Dimitrios Georgakopoulos, Massimo Villari, and Rajiv Ranjan. 2017. Deep osmosis: Holistic distributed deep learning in osmotic computing. *IEEE Cloud Computing* 4, 6 (2017), 22–32.

[21] Jihong Park, Sumudu Samarakoon, Mehdi Bennis, and Mérouane Debbah. 2018. Wireless network intelligence at the edge. *arXiv preprint arXiv:1812.02858* (2018).

[22] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. (2017).

[23] Dhananjay S Phatak and Israel Koren. 1995. Complete and partial fault tolerance of feedforward neural nets. *IEEE Transactions on Neural Networks* 6, 2 (1995), 446–456.

[24] G. Roig, X. Boix, H. Ben Shitrit, and P. Fua. 2011. Conditional Random Fields for multi-camera object detection. In *2011 International Conference on Computer Vision*. 563–570.

[25] Carlo H Sequin and RD Clay. 1990. Fault tolerance in artificial neural networks. In *1990 IJCNN international joint conference on neural networks*. IEEE, 703–708.

[26] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[27] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Training very deep networks. In *Advances in neural information processing systems (NeurIPS)*. 2377–2385.

[28] Zeyi Tao and Qun Li. 2018. eSGD: Communication Efficient Distributed Deep Learning on the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotedge18/presentation/tao

[29] Surat Teerapittayanon, Bradley McDanel, and HT Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 328–339.

[30] Cesar Torres-Huitzil and Bernard Girau. 2017. Fault and error tolerance in neural networks: A review. *IEEE Access* 5 (2017), 17322–17341.

[31] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. 2019. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1205–1221.

[32] Shiqiang Wang, Murtaza Zafer, and Kin K Leung. 2017. Online placement of multi-component applications in edge computing environments. *IEEE Access* 5 (2017), 2514–2533.

[33] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. 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, 4.

[34] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. 2019. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture* 98 (2019), 289 – 330.

[35] Li Zhou, Hao Wen, Radu Teodorescu, and David HC Du. 2019. Distributing Deep Neural Networks with Containerized Partitions at the Edge. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[36] Zhi-Hua Zhou and Shi-Fu Chen. 2003. Evolving fault-tolerant neural networks. *Neural Computing & Applications* 11, 3-4 (2003), 156–160.