Journal of Signal Processing Systems (2023) 95:1355–1370
https://doi.org/10.1007/s11265-023-01852-0

Timing Performance Benchmarking of Out-of-Distribution Detection Algorithms

Siyu Luan[1] · Zonghua Gu[1] · Amin Saremi[1] · Leonid Freidovich[1] · Lili Jiang[2] · Shaohua Wan[3]

Received: 13 December 2022 / Revised: 18 January 2023 / Accepted: 28 January 2023 / Published online: 23 March 2023
© The Author(s) 2023

Abstract
In an open world with a long-tail distribution of input samples, Deep Neural Networks (DNNs) may make unpredictable mistakes for Out-of-Distribution (OOD) inputs at test time, despite high levels of accuracy obtained during model training. OOD detection can be an effective runtime assurance mechanism for safe deployment of machine learning algorithms in safety–critical applications such as medical imaging and autonomous driving. A large number of OOD detection algorithms have been proposed in recent years, with a wide range of performance metrics in terms of accuracy and execution time. For real-time safety–critical applications, e.g., autonomous driving, timing performance is of great importance in addition to accuracy. We perform a comprehensive and systematic benchmark study of multiple OOD detection algorithms in terms of both accuracy and execution time on different hardware platforms, including a powerful workstation and a resource-constrained embedded device, equipped with both CPU and GPU. We also profile and analyze the internal details of each algorithm to identify the performance bottlenecks and potential for GPU acceleration. This paper aims to provide a useful reference for the practical deployment of OOD detection algorithms for real-time safety–critical applications.

Keywords Machine Learning · Deep Learning · Out-of-Distribution detection · Real-time systems · Embedded systems

1 Introduction

Deep Neural Networks (DNNs) trained with Deep Learning are widely used in many application domains today, including safety–critical autonomous systems, e.g., such as Autonomous Vehicles (AVs), esp. for environment perception. The high complexity of modern DNNs causes them to be blackbox-like with little insight of their internal workings, and their complexity and opaqueness pose significant challenges to high levels of safety certification with traditional Verification and Validation techniques. Furthermore, large DNNs often lack strong generalization capability beyond the training data distribution. Consider the perception system of an AV, which faces an open world with a long-tail distribution of input samples that may be Out-of-Distribution (OOD), i.e., the training and testing data may not be i.i.d (independent and identically distributed), and a test data sample at runtime may fall outside of the statistical distribution of the training dataset. Such distribution shifts may cause sharp drops in the classification accuracy of DNNs, as the typical softmax-based classifier often gives incorrect yet over-confident predictions for OOD inputs. A well-trained DNN may achieve high accuracy for In-Distribution (ID) inputs, but may fail catastrophically when faced with OOD inputs, with potentially serious safety consequences. One solution is to develop accurate OOD detection algorithms [1] as part of a runtime assurance architecture to achieve high levels of safety certification for autonomous systems with DNN-based perception systems. Instead of blindly trusting the prediction results, the perception system equipped with an OOD detector should "fail loudly" by declaring "I don't know" upon encountering OOD inputs, so the system knows that the DNN's prediction result can no longer be

✉ Zonghua Gu
zonghua.gu@umu.se

✉ Shaohua Wan
shaohua.wan@ieee.org

1 Department of Applied Physics and Electronics, Umeå University, Umeå, Sweden

2 Department of Computing Science, Umeå University, Umeå, Sweden

3 Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China

trusted. The higher-level safety supervision system may subsequently take corrective measures to ensure safety.

A large number of different OOD detection algorithms have been proposed in recent years. Some benchmarking studies exist that focus on the accuracy of OOD detection algorithms [2]. For real-time safety–critical applications, e.g., autonomous driving, timing performance is of great importance in addition to accuracy. In this paper, we perform a comprehensive and systematic benchmark study of multiple OOD detection algorithms in terms of both accuracy and execution time on different hardware platforms, including a powerful workstation and a resource-constrained embedded device, equipped with both CPU and GPU. We also profile and analyze the internal details of each algorithm to identify the performance bottlenecks and potential for GPU acceleration. This paper aims to provide a reference for the practical deployment of OOD detection algorithms for real-time safety–critical applications, to help the designer choose the most appropriate OOD detection algorithm based on application requirements and hardware platform capability.

This paper is structured as follows: we present the background and related work in Section 2; the experimental setup in Section 3; results and analysis in Section 4; and finally, conclusions and future work in Section 5.

2 Background and Related Work

Let $\mathcal{X}$ be the input space and $\mathcal{Y}$ the label space. A parametric machine learning model, e.g., a DNN, is defined as a mapping function $f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$ from the input space to the label space with learnable parameters θ. Given a set of n training samples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, which are drawn from the training distribution $P_{tr}(X, Y)$, the supervised learning problem is to find an optimal model that can generalize best on data drawn from test distribution $P_{te}(X, Y)$:

$$f_\theta^* = \mathrm{argmin}_{f_\theta} \mathbb{E}_{(X,Y)\sim P_{te}} \left[l\left(f_\theta(X), Y\right) \right] \tag{1}$$

The conventional machine learning assumption is that the training samples and test samples are both i.i.d. realizations from a common underlying distribution, that is, $P_{tr}(X, Y) = P_{te}(X, Y)$. Given this assumption, Empirical Risk Minimization (ERM), which minimizes the average loss on training samples $\mathcal{L}_{ERM} = \frac{1}{n}\sum_i^n l\left(f_\theta\left(x_i\right), y_i\right)$, can be used to train a machine learning model that can generalize well to the test distribution.

However, in real application scenarios, the test distribution may shift/deviate from the training distribution, that is, $P_{tr}(X, Y) \neq P_{te}(X, Y)$. The distribution shift may be due to different reasons, such as dataset sample selection bias, changes in the natural environment, or even adversarial attacks. The problem of *OOD detection* is detecting when a test input is sampled from $P_{te}(X, Y)$ which is different from the training distribution $P_{tr}(X, Y)$. (The related problem of *OOD generalization* [3] is defined as the instantiation of supervised learning problem where the test distribution $P_{te}(X, Y)$ shifts from the training distribution $P_{tr}(X, Y)$ and remains unknown during the training phase.) For a given input x and model f_θ, an OOD detection algorithm is a binary classifier that computes an OOD score (or anomaly score) $S\left(x, f_\theta\right)$ and compares it to a given threshold λ, i.e., input x is OOD if $S\left(x, f_\theta\right) \geq \lambda$. The threshold λ is typically chosen so that a large fraction of ID data (e.g., 95%) is correctly classified. A threshold-independent metric, e.g., AUROC (Area Under the Receiver Operating Characteristics curve), can be used to remove the dependence on the threshold *th* for evaluating classification accuracy.

A large number of OOD detection algorithms have been developed in recent years, with different OOD score functions. Most authors consider the detection accuracy, but few studies focus on the issue of execution time, which is important for real-time embedded systems with limited computing and memory resources [4, 5]. Yang et al. [2] present a unified codebase called OpenOOD for comprehensive benchmarking of different OOD detection algorithms. They can be categorized into classification-based, density-based, distance-based, and reconstruction-based methods. In this paper, we perform timing performance measurements for 13 post-hoc OOD detection algorithms, including:

1. Classification-based: MSP, ODIN, MLS, KL-Matching, VIM, GradNorm, DICE, ReAct.
2. Density-based: EBO, OpenMax.
3. Distance-based: MDS, KNN, Gram.

We focus on the *post-hoc* OOD detection algorithms that work with pre-trained DNNs, since they are more suitable for real-world practice without the expensive training process and/or need for OOD samples in other approaches, e.g., training-time regularization and training with outlier exposure [2]. We do not consider reconstruction-based methods based on generative models, e.g., Generative Adversarial Networks (GANs) [6], Autoencoders (AEs) incl. Variational Autoencoders (VAEs) [7], which rely on the discrepancy of reconstruction errors between ID and OOD input samples, as their accuracy and execution time metrics are dependent on the architecture and size of the generative model, hence difficult to quantify and compare with the other methods in a general sense. Similarly, we do not consider distance-based methods based on monitoring hidden layers with outlier detection algorithms [8, 9], since their accuracy and execution time metrics are dependent on the number of monitored layers and (highly-configurable) size/complexity of the outlier detection method.

Below is a brief introduction to the selected OOD algorithms.

1. Hendrycks and Gimpel [10] propose Maximum Softmax Probability (**MSP**), based on the idea that the DNN softmax score reflects the confidence of the DNN model on the test input. The higher the MSP (the highest softmax score among all classes), the more likely the test data is ID, and vice versa. Therefore, we can use a threshold of softmax score to determine whether the current test input is OOD. MSP is widely used as the comparison baseline method due to its simplicity [11], but it may not be very accurate, since a DNN often outputs incorrect yet overconfident predictions for OOD data, i.e., the ranges of softmax scores of ID and OOD data often overlap with each other.
2. Liang et al. [12] propose **ODIN**, which uses softmax with temperature as confidence on perturbed inputs, based on the observation that temperature scaling and adding small perturbations to the input image can separate the softmax score distributions between ID and OOD inputs, allowing for more effective OOD detection.
3. Hendrycks et al. [13] propose MaxLogit (**MLS**), which uses the negative of the maximum unnormalized logit for an anomaly score $-\max_k f(x)_k$. Unlike the softmax scores that are normalized to sum to 1, the logits are unnormalized, hence they are not affected by the number of classes and can serve as a better comparison baseline than MSP [10] for OOD detection.
4. Hendrycks et al. [13] propose **KL-Matching**, which works by capturing the typical shape of each class's posterior distribution and forming posterior distribution templates for each class. During test time, the network's softmax posterior distribution is compared to these templates to compute the anomaly score.
5. Wang et al. [14] propose Virtual-logit Matching (**VIM**), which combines the class-agnostic score from feature space and the ID class-dependent logits. An additional logit representing the virtual OOD class is generated from the residual of the feature against the principal space and then matched with the original logits by constant scaling. The probability of this virtual logit after softmax is the indicator of OOD-ness.
6. Huang et al. [15] propose **GradNorm**, using information extracted from the gradient space. GradNorm directly employs the vector norm of gradients, backpropagated from the Kullback–Leibler (KL) divergence between the softmax score and a uniform probability distribution. The key idea is that the magnitude of gradients is higher for ID data than that for OOD data, making it informative for OOD detection.
7. Sun et al. [16] propose Directed Sparsification (**DICE**). The key idea is to rank weights based on a measure of contribution, and selectively use the most salient weights to derive the output for OOD detection, based on the insight that reliance on unimportant weights and units can attribute to the brittleness of OOD detection.
8. Sun et al. [17] propose **ReAct**, which works by clipping the activation value that exceeds a certain threshold, and then keeping the activations within a certain range based on the observation that abnormally high activations on OOD data can harm their detection.
9. Liu et al. [18] propose Energy-Based OOD detection (**EBO**) using energy scores. Unlike softmax confidence scores, energy scores are theoretically aligned with the probability density of the inputs and are less susceptible to the overconfidence issue. Within this framework, energy can be flexibly used as a scoring function for any pre-trained DNN as well as a trainable cost function to shape the energy surface explicitly for OOD detection.
10. Bendale and Boult [19] propose a new model layer **OpenMax**, which estimates the probability of an input being from an unknown class. A key element of estimating the unknown probability is adapting Meta-Recognition concepts to the activation patterns in the penultimate layer of the network, to reject unrelated open set images.
11. Lee et al. [20] propose the Mahalanobis Distance (**MDS**) algorithm. It computes layer-wise Mahalanobis distances from class-conditional feature distributions, which are used to train a Logistic Regression classifier.
12. Sun et al. [21] propose K-Nearest Neighbor (**KNN**)-based OOD detection, which computes the k-th nearest neighbor (KNN) distance between the embeddings of the test input and the embeddings of the training set and compares it to a threshold.
13. Sastry et al. [22] propose **Gram**, by computing Gram Matrices at every layer and checking for anomalously high or low values by comparing each value with its respective range observed over the training data.

3 Experimental Setup

We consider two well-known benchmark datasets, namely Mnist [23] and Cifar10 [24]. We consider two types of OOD datasets: Near-OOD datasets only have semantic shift compared with ID datasets, while Far-OOD datasets further contain obvious covariate (domain) shift [2]. In the Mnist benchmark, the ID dataset is the Mnist dataset, the Near-OOD datasets include NotMnist [25], FashionMnist [26], and the Far-OOD datasets include Texture [27], Cifar10 [24], TinyImageNet [28], and Places365 [29], respectively. The DNN architecture is LeNet [30]. For all datasets, the input image is resized to the uniform size of $28 \times 28 \times 3$. In

the Cifar10 benchmark, the ID dataset is the Cifar10 dataset, the Near-OOD datasets include Cifar100 [31], TinyImageNet, and the Far-OOD datasets include Mnist, SVHN [32], Texture, and Places365, respectively. The DNN architecture is ResNet-18 [33]. For all datasets, the input image is resized to the uniform size of $32 \times 32 \times 3$. Table 1 shows the dataset details, including the dataset names and their sizes (number of data items in the dataset).

We consider two hardware platforms, one powerful deep learning workstation platform, and one resource-constrained embedded device (Jetson Nano 2 GB). Their technical specifications are shown in Table 2. We use the following abbreviations: **JSNCPU** for Jetson Nano CPU; **JSNGPU** for Jetson Nano GPU; **WSCPU** for Workstation CPU; **WSGPU** for Workstation GPU. In our measurements, we only use one GPU on the workstation even though it is equipped with two GPUs. The Deep Learning workstation has separate CPU memory of size 125 GB and GPU memory of size 11 GB, whereas Jetson Nano has a unified memory of size 2 GB, shared between the CPU and the GPU. We adopt batchsize 128 for the Mnist benchmark, and 32 for the Cfar10 benchmark, based on the maximum batchsize that can fit on the Jetson Nano device. For each benchmark, we measure the total execution time of the entire testset and divide it by the testset size to obtain the execution time of each instance, as reported in this paper.

We consider two performance metrics: for OOD detection accuracy, we consider the AUROC; for the execution time metric, we measure the average execution time for one input sample on either CPU or GPU, obtained by dividing the total execution time of the entire test dataset by the number of data items in the dataset. Common to all OOD detection algorithms is one forward inference for the DNN to obtain the feature $f(x)$ (the logits) in the penultimate layer, and then compute either the softmax scores with a softmax layer, or a one-hot prediction result with a hard max function over the logits. In Section 4.1, we measure the total execution time, which includes both the DNN forward inference time to compute $f(x)$, and the OOD detection time, which includes the final softmax or hard max function as part of it. (The most time-consuming part of the DNN forward inference is the convolutional layers plus zero or more fully-connected layers to compute $f(x)$. Since the final softmax or hard max function consumes a tiny fraction of the total inference time, and it is often inter-mixed within the OOD detection code, we count it as part of the OOD detection time.) The latency due to OOD detection is computed by subtracting one forward inference time from the total execution time. In Section 4.2, we report the execution time percentage of OOD detection, defined in Eq. (2):

$$\text{OOD}_{\text{percent}} = \frac{(\text{Total time} - \text{one DNN inference time})}{\text{Total time}} \quad (2)$$

We control a code section to run on the CPU or GPU by placing the relevant tensor data structures on the CPU or GPU with calls to `torch.Tensor.cpu()` and `torch.Tensor.cuda()`, respectively. While DNN forward inference can run on either the CPU or the GPU, the OOD detection code can always be on the CPU, but not always on the GPU, since many OOD detection algorithms invoke APIs defined in the NumPy library [34] or the Scikit-learn library [35] that may or may not be supported on the GPU. When we denote a timing measurement on JSNGPU or WSGPU, the implication is that the DNN forward inference always runs on the GPU, whereas the OOD detection code runs on the GPU whenever possible for maximum acceleration, but sometimes it must run on the CPU due to lack of GPU support in Scikit-learn. In other words, we can always change any statement `torch.Tensor.cuda()` into `torch.Tensor.cpu()`, but not vice versa. If DNN forward inference and OOD detection are run on different processing units (GPU and CPU, respectively), then the OOD detection time measured on the workstation may include additional memory transfer delays between the separate CPU memory and GPU memory, which are typically quite short for our use case of OOD detection, since the transferred data sizes are not very large. In our experiments, for timing measurements on the GPU, we use the original source code in OpenOOD, where the DNN forward inference step always runs on the GPU, and the OOD detection algorithm may run (partially or fully) on the GPU; for timing measurements on the CPU, we replace all calls to `torch.Tensor.cuda()` with `torch.Tensor.cpu()`.

Table 1 ID and OOD datasets for the two benchmarks with Mnist or Cifar10 as the ID dataset.

Benchmark	Type	Dataset name	Dataset size
Mnist	ID	Mnist	9004
	Near-OOD	NotMnist	17,724
		FashionMnist	10,000
	Far-OOD	Texture	5640
		Cifar10	10,000
		Tin	10,000
		Places365	36,500
Cifar10	ID	Cifar10	9000
	Near-OOD	Cifar100	9000
		Tin	8793
	Far-OOD	Mnist	70,000
		SVHN	26,032
		Texture	5640
		Places365	35,195

Table 2 Hardware Configuration of Two Platforms.

Deep Learning Workstation	CPU: AMD Ryzen Threadripper 2990WX 32-Core Processor. Clock speed: 3.0 GHz
	GPU: 2×GeForce RTX 2080 Ti GPU
	CPU memory: 125 GB, GPU memory: 11 GB
NVIDIA Jetson Nano 2 GB	CPU: Quad-core ARM A57 Processor. Clock speed: 1.43 GHz
	GPU: 128-core Maxwell GPU
	Unified memory: 2 GB

For a code section that runs on the CPU, we use the following code fragment to measure its execution time [36]:

```
import time as timer
start = timer.time()
# Code section to be measured
# (timer.time() - start) is the measured
execution time on the CPU (in seconds), to be
saved later.
```

For a code section that runs on the GPU, we use the following code fragment to measure its execution time [36]:

```
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

start.record()
# Code section to be measured
end.record()

# Waits for the code section to finish
execution
torch.cuda.synchronize()

# start.elapsed_time(end)is the measured
execution time on the GPU (in ms), to be saved
later.
```

Since cuda operations are asynchronous and return immediately, the code uses a combination of `torch.cuda.Event()`, a synchronization marker, and `torch.cuda.synchronize()`, a directive for waiting for the event to complete. Based on documentation of PyTorch, `torch.cuda.synchronize()` "waits until the completion of all work currently captured in this event. This prevents the CPU thread from proceeding until the event completes."

If a code section contains a mixture of code that runs on both CPU and GPU, we use `timer.time()` to measure its total execution time, while adding `torch.cuda.synchronize()` just before the second call to `timer.time()`, to wait for the completion of all the code executions on the GPU before calculating the time interval length. This situation is quite common in OOD detection algorithms, as some parts of the algorithm must run on the CPU due to lack of library support on the GPU, while some parts may be run on the GPU for maximum acceleration.

4 Results and Analysis

In this section, we present execution time measurements in Section 4.1; execution time percentage of OOD detection in Section 4.2; detailed measurements of different components of OOD detection algorithms in Section 4.3; and AUROC vs. execution time in Section 4.4.

4.1 Execution Time Measurements

4.1.1 Mnist as ID Dataset

Figure 1(a), (b) show the results on JSNCPU and JSNGPU, respectively (note the different timescale on the y-axis, since GPU is much faster than CPU). We can see that MDS, Gram, and ODIN are the most time-consuming on the CPU. Gram is time-consuming since it involves multiplication operations of higher-order Gram matrices. MDS and ODIN both incur two DNN forward inferences and one back-propagation of gradients for input preprocessing of adding a small perturbation to the input image. Back-propagation of gradients is quite time-consuming, esp. on the CPU (roughly twice the time cost of one forward inference), which explains their high execution times. GPU helps achieve significant speedups for MDS and ODIN relative to running on the CPU, esp. thanks to the speedup of the back-propagation step, hence they are no longer in the top three most time-consuming algorithms on the GPU. Instead, Gram, GradNorm and KNN are the most time-consuming algorithms on the GPU.

Figure 1(c) shows the results on WSCPU. The most time-consuming algorithms are still MDS, ODIN, and Gram, but the relative differences from the other algorithms are less pronounced. Figure 1(d) shows the results on WSGPU, which show similar trends as the case of the JSNGPU.

4.1.2 Cifar10 as ID Dataset

From Fig. 2(a), (c), we can see that MDS and ODIN are the most time-consuming on both JSNCPU and WSCPU: their execution times are increased significantly compared to the Mnist benchmark due to the increase in the input image size and the more complex DNN model architecture in the

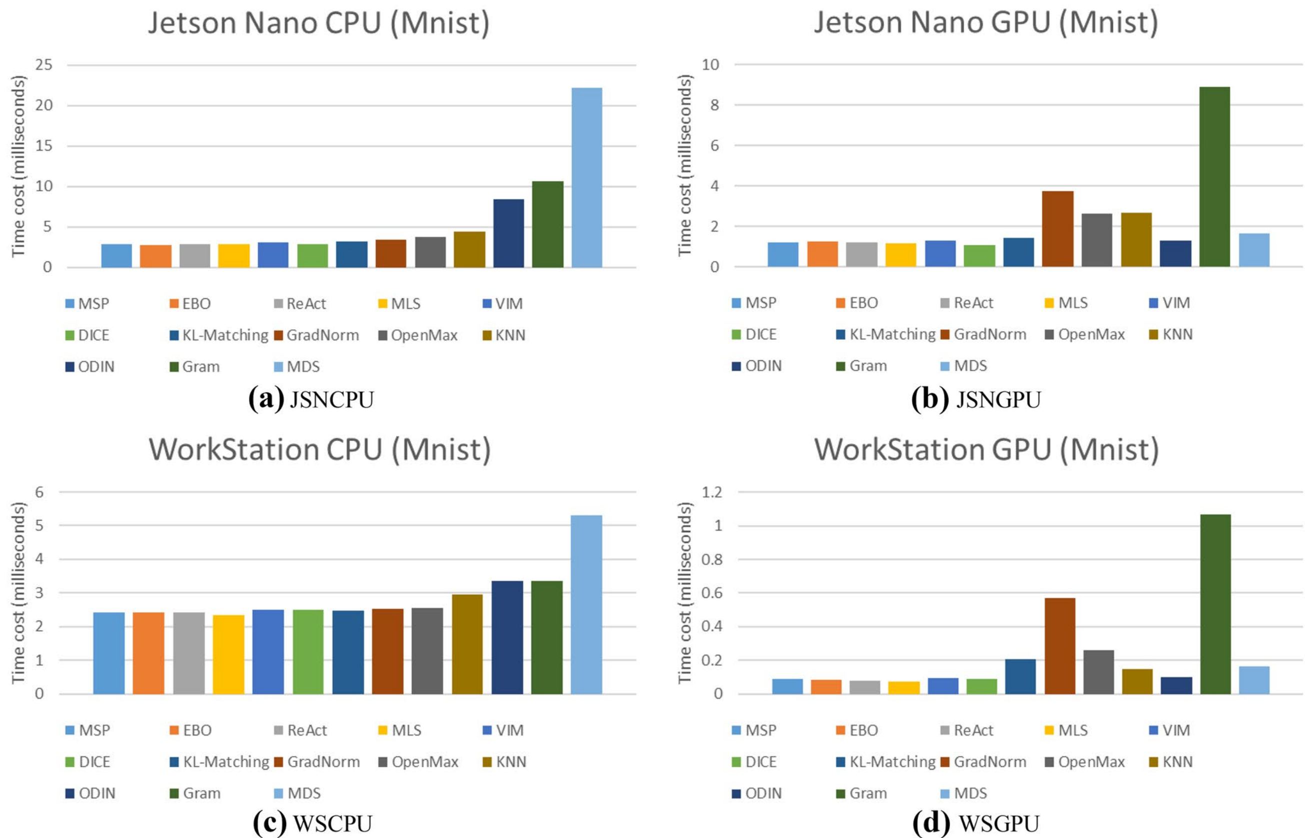


Figure 1 Average execution time with Mnist as the ID dataset.

Cifar10 benchmark. The other algorithms have similar execution times. (Gram runs out of memory on Jetson Nano due to the larger input image size (32 × 32x3) and larger DNN architecture (ResNet-18) in the Cifar10 benchmark relative to the Mnist benchmark (28 × 28x3, LeNet), for both JSNCPU and JSNGPU, hence it is not shown in Fig. 2(a), (b)).

From Fig. 2(b), (d), we can see that the differences among different algorithms become more pronounced on the GPU than on the CPU. MDS and ODIN are the most time-consuming on the JSNGPU, whereas Gram, MDS, and KNN are the most time-consuming on WSGPU.

4.2 Execution Time Percentage of OOD Detection

In this section, we report the execution time percentage of OOD detection as defined in Eq. (2).

4.2.1 Mnist as ID Dataset

Figure 3 shows the results for the Mnist benchmark. From Fig. 3(a), we can see that Gram has the highest percentage of 78.9%, followed by ODIN, MDS and KNN on JSNCPU. Figure 3(c) shows similar trends on WSCPU, with minor differences from Fig. 3(a). The percentages for the other algorithms are quite low, indicating their relative computation efficiency.

Figure 3(b), (d) shows the results on JSNGPU and WSGPU. Since GPU helps achieve a significant speedup of the DNN forward inference step, but not for most OOD detection algorithms, which may run on the CPU (partially or fully). Hence OOD detection accounts for a significantly larger proportion of the total execution time compared to CPU execution for most algorithms, reaching close to 100% in some cases (e.g., Gram). This highlights the importance of optimizing the computation efficiency of OOD detection algorithms, esp. for smaller DNNs with fast inference times.

4.2.2 Cifar10 as ID Dataset

Figure 4 shows the results for the Cifar10 benchmark. Compared to Fig. 3 for the Mnist benchmark, the execution time percentages of most OOD detection algorithms on both GPU and GPU are reduced compared to the Mnist benchmark, since the DNN forward inference time of ResNet-18 is much larger than that of LeNet. The reduction effect is less pronounced for ODIN and MDS, since they both incur two

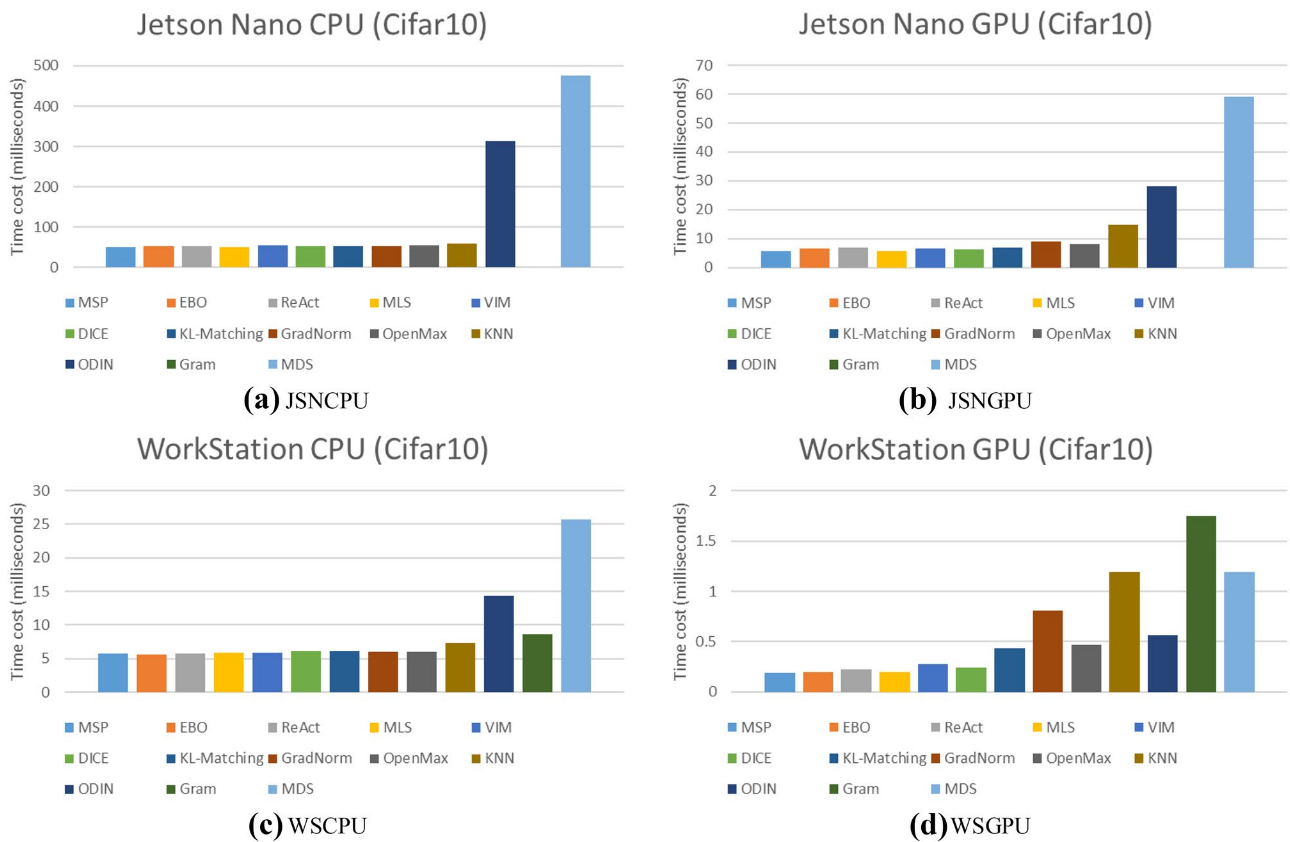


Figure 2 Average execution time with Cifar10 as the ID dataset. (Gram is not shown in (**a**), (**b**), since it runs out of memory on Jetson Nano).

DNN forward inferences and one back-propagation of gradients, which are all accelerated by adopting a more powerful CPU or switching from CPU to GPU.

4.3 Time Measurements of Different Components of OOD Detection Algorithms

In this section, we use the Cifar10 benchmark to further investigate the effect of GPU acceleration on the different components of seven OOD detection algorithms that are more time-consuming, including ODIN, KL-Matching, GradNorm, OpenMax, MDS, KNN, and Gram. The other algorithms are quite simple and efficient, according to our measurement results in Sections 4.1 and 4.2, hence not considered in this section. We do not consider the Mnist benchmark, since the LeNet architecture for the Mnist benchmark is quite small and is less practically relevant compared to the ResNet-18 architecture for the Cifar10 benchmark.

As mentioned in Section 3, for timing measurements on the GPU, the DNN forward inference step always runs on the GPU, and the OOD detection algorithm may run (partially or fully) on the GPU. In this section, in the columns marked "JSNGPU" and "WSGPU" of the tables containing timing measurement results: if an algorithm component runs fully on the GPU, then we do not add any annotation; if an algorithm component runs fully on the CPU, then we add the annotation "(on CPU)" next to its execution time measurement; if an algorithm component contains a mixture of code that runs on both CPU and GPU, then we add the annotation "(on CPU/GPU)".

4.3.1 ODIN

ODIN (Liang et al. [12]) improves upon MSP [10] based on the observation that using temperature scaling and input preprocessing of adding small perturbations to the input can help separate the softmax score distributions between ID and OOD inputs, allowing for more effective detection. ODIN includes three steps: one DNN forward inference (Inf); one back-propagation of gradients to apply a small perturbation to the input (Backprop) as input preprocessing; and finally another DNN forward inference (Inf) to compute the MSP of the perturbed input.

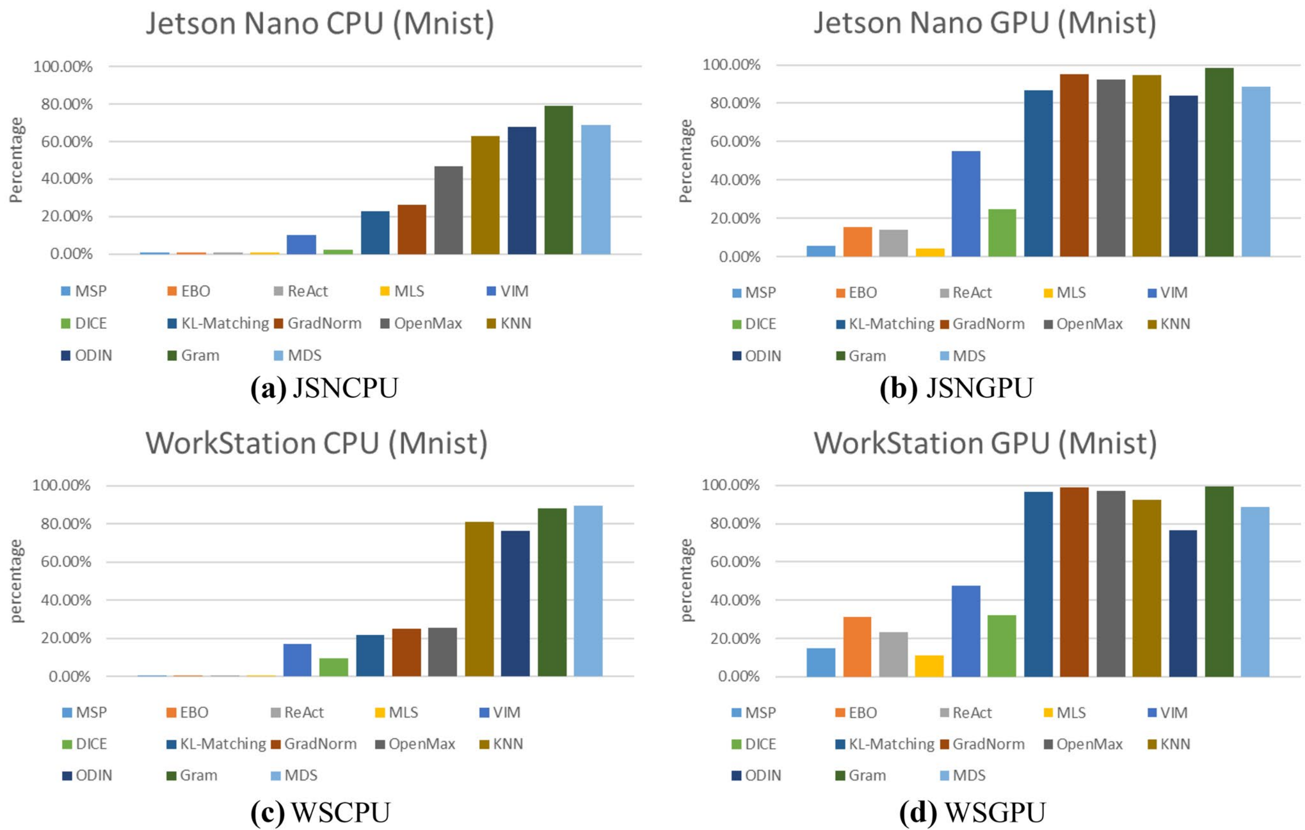


Figure 3 Execution time percentage of OOD for the Mnist benchmark.

Step 1: For input x, perform one DNN forward inference to compute the softmax score for each class with temperature scaling:

$$S_i(x;T) = \frac{\exp\left(\frac{f_i(x)}{T}\right)}{\sum_{j=1}^{N}\exp\left(\frac{f_j(x)}{T}\right)} \tag{3}$$

where N is the number of classes; $f_i(x)$ is the logit value for class i.
Step 2: Preprocess the input by applying a small perturbation to it, computed by back-propagating the gradient of the cross-entropy loss w.r.t the input:

$$\widetilde{x} = x - \varepsilon \mathrm{sign}\left(-\nabla_x \log S_{\hat{y}}(x;T)\right) \tag{4}$$

where the parameter ε is the perturbation magnitude.
Step 3: For the preprocessed input $\widetilde{x}$, perform another DNN forward inference to compute the softmax score for each class $S_i(\widetilde{x};T)$, and compare the MSP to a threshold δ. The input x is classified as OOD if the MSP is less than δ, which indicates that the DNN is not confident about the classification result.

From Table 3, we can see that on both Jetson Nano and Workstation platforms, GPU achieves significant acceleration for all three steps, including two (DNN forward inference plus softmax) steps and one back-propagation step.

4.3.2 KL-Matching

KL-Matching (Hendrycks et al. [13]) captures the typical shape of each class's posterior distribution and forms posterior distribution templates for each class. During test time, the network's softmax posterior distribution is compared to these templates to compute the anomaly score. More concretely, we compute k different distributions d_k, one for each class, on the validation dataset $\mathcal{X}_{val}$, denoting the posterior distribution template for class k.

$$d_k = \mathbb{E}_{x' \sim \mathcal{X}_{val}}[p(y|x')], \tag{5}$$

where $k = \mathrm{argmax}_k\, p(y = k|x')$

For each test input x, we perform forward inference (Inf) to obtain the softmax scores $p(y|x)$, then compute the anomaly

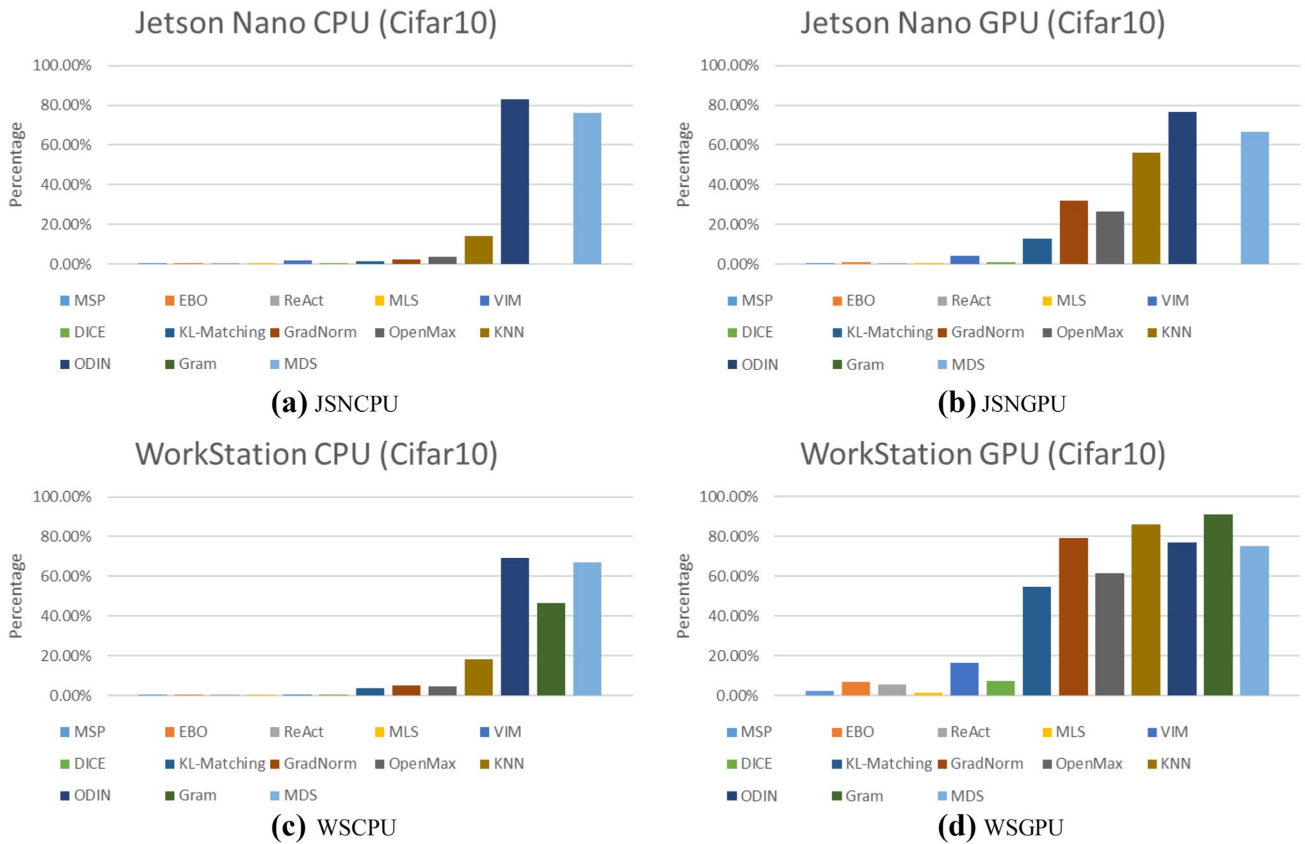


Figure 4 Execution time percentage of OOD detection for the Cifar10 benchmark. (Gram is not shown in (**a**), (**b**), since it runs out of memory on Jetson Nano).

score as the minimum KL Divergence (KLD) among all classes $\min_k \mathrm{KL}[p(y|x)||d_k]$, i.e., KLD to the posterior distribution template d_k of the class k that is closest to $p(y|x)$.

Unlike ODIN, which can execute entirely on the CPU or entirely on the GPU, KL-Matching must run on the CPU since uses some Scikit-learn functions not supported by the GPU (specifically, the function `pairwise_distances_argmin_min()`). From Table 4, we can see that the KLD step has a slightly longer execution time when the Inf step is run on the GPU (in columns JSNGPU and WSGPU) due to switching from GPU execution to CPU execution. But the calculation in the KLD step is relatively simple and takes little time, and running Inf on the GPU helps to significantly reduce the overall execution time compared to running it on the CPU on each platform.

Table 3 Execution time of different components of ODIN (time unit: ms).

ODIN	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf + softmax	52.92	5.89	4.19	0.12
Backprop	195.98	13.30	6.36	0.28
Inf + softmax	58.14	5.95	3.18	0.14

4.3.3 GradNorm

GradNorm (Huang et al. [15]) works as follows: we first perform DNN forward inference, then compute the gradients w.r.t. each parameter by backpropagating the KL divergence between the softmax score and a uniform distribution $\mathbf{u} = [1/C, 1/C, ..., 1/C]$. The predictive probability distribution is the softmax score. The KL divergence for backpropagation is:

$$\mathrm{D_{KL}}(\mathbf{u}||\mathrm{softmax}(\mathrm{f}(\mathrm{x}))) = -\frac{1}{C}\sum_{c=1}^{C}\log\frac{e^{\frac{f_c(x)}{T}}}{\sum_{j=1}^{C}e^{\frac{f_j(x)}{T}}} - \mathrm{H}(\mathbf{u}) \quad (6)$$

where the first term is the cross-entropy loss between the softmax score distribution and the uniform distribution **u**,

Table 4 Execution time of different components of KL Matching (time unit: ms).

KL-Matching	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf	51.82	5.90	5.60	0.14
KLD	0.80	0.87 (on CPU)	0.16	0.17 (on CPU)

Table 5 Execution time of different components of GradNorm (time unit: ms).

GradNorm	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf	50.89	5.89	5.64	0.14
GradNorm	1.15	2.78 (on CPU/GPU)	0.29	0.55 (on CPU/GPU)

and the second term H(**u**) is the entropy of **u** (a constant). The KL divergence measures how much the predictive distribution deviates from the uniform distribution. ID data is expected to have a larger KL divergence because the prediction tends to concentrate on the ground-truth class and is thus distributed less uniformly.

For a given parameter w, the gradient of the above KL divergence is:

$$\frac{\partial}{\partial w} D_{KL}(u||\mathrm{softmax}(f(x))) = \frac{1}{C}\sum_{i=1}^{C}\frac{\partial L_{CE}(f(x),\, i)}{\partial w} \tag{7}$$

where L_{CE} is the Cross-Entropy loss for classification.

The Gradient Norm (GradNorm) is defined via a vector norm of gradients of the selected parameters:

$$\mathrm{S(x)} = \left\|\frac{\partial}{\partial \mathbf{w}} D_{KL}(u||\mathrm{softmax}(f(x)))\right\|_p \tag{8}$$

where $\|\|_p$ denotes the L_p-norm, and **w** is the set of parameters in vector form. The GradNorm is higher for ID inputs than for OOD inputs.

When the Inf step runs on the GPU. the implementation of GradNorm consists of a mixture of CPU execution and GPU execution steps. From Table 5, we can see that the GradNorm step has a slightly longer execution time when the Inf step is run on the GPU (in columns JSNGPU and WSGPU) due to switching between GPU execution and CPU execution, but the overall execution time is still reduced significantly on the GPU compared to the CPU.

4.3.4 OpenMax

The OpenMax algorithm (Bendale and Boult [19]) replaces the conventional softmax layer with the OpenMax layer. We first perform DNN forward inference (Inf) until the penultimate layer. In the OpenMax layer, we use the Weibull CDF probability on the distance between x and μ_i for the core of the rejection estimation. The model μ_i is computed using the images associated with category i, images that were classified correctly (top-1) during the training process. We compute weights for the α largest activation classes and use it to scale the Weibull CDF probability. We then compute a revised activation vector with the top scores changed. We compute a pseudo-activation for the unknown unknown class with index 0, keeping the total activation level constant. Including the unknown unknown class, the OpenMax probabilities are computed with the revised activation vector.

From Table 6, we can draw similar conclusions as KL-Matching and GradNorm.

4.3.5 MDS

MDS (Lee et al. [20]) is a parametric distance-based method that works as follows: in the training phase, we obtain a generative classifier, assuming that each class-conditional distribution follows the multivariate Gaussian distribution in the feature space. Specifically, we define C class-conditional Gaussian distributions with a tied covariance $\sum : P(f(x)|y=c) = \mathcal{N}(f(x)|\mu_c, \sum)$ where μ_c is the mean of multivariate Gaussian distribution of class $c\epsilon\{1,\ldots,C\}$. To estimate the parameters of the generative classifier from the pre-trained softmax classifier, we compute the empirical class mean and covariance of training samples $\{(x_1,y_1),\ldots,(x_N,y_N)\}$:

$$\hat{\mu}_c = \frac{1}{N_c}\sum_{i:y_i=c} f(x_i),$$

$$\widehat{\sum} = \frac{1}{N}\sum_c \sum_{i:y_i=c} \left(f(x_i) - \hat{\mu}_c\right)\left(f(x_i) - \hat{\mu}_c\right)^{\mathrm{T}}$$

where N_c is the number of training samples with label c.

During inference, for each layer $l \in \{1,\ldots,L\}$, three steps are run, including one DNN forward inference (Inf); one back-propagation of gradients to apply a small perturbation to the input (Backprop); and finally another DNN forward inference (Inf) to compute the confidence score.

Step 1: Find the closest class as measured by Mahalanobis distance (Inf and M-Dis):

$$\hat{c} = \mathrm{argmin}_c\left(f_l(x) - \hat{\mu}_{l,c}\right)^{\mathrm{T}} \widehat{\sum}_l^{-1}\left(f_l(x) - \hat{\mu}_{l,c}\right) \tag{9}$$

Step 2: Apply a small perturbation to the test input (Backprop):

Table 6 Execution time of different components of OpenMax (time unit: ms).

OpenMax	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf	53.82	5.50	5.64	0.14
OpenMax	1.94	1.97 (on CPU/GPU)	0.28	0.22 (on CPU/GPU)

Table 7 Execution time of different components of MDS (time unit: ms).

MDS	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf	147.16	17.69	10.30	0.35
M-Dis	2.07	0.68 (on CPU/GPU)	0.19	0.09 (on CPU/GPU)
Backprop	149.73	8.68	4.69	0.29
Inf	173.67	18.10	9.44	0.39
M-Dis	0.33	0.70 (on CPU/GPU)	0.11	0.09 (on CPU/GPU)

$$\hat{x} = x - \varepsilon \mathrm{sign}(\nabla_x (f_l(x) - \hat{\mu}_{l,\hat{c}})^{\mathrm{T}} \hat{\Sigma}_l^{-1} (f_l(x) - \hat{\mu}_{l,\hat{c}})) \quad (10)$$

Step 3: Compute confidence score (Inf and M-Dis):

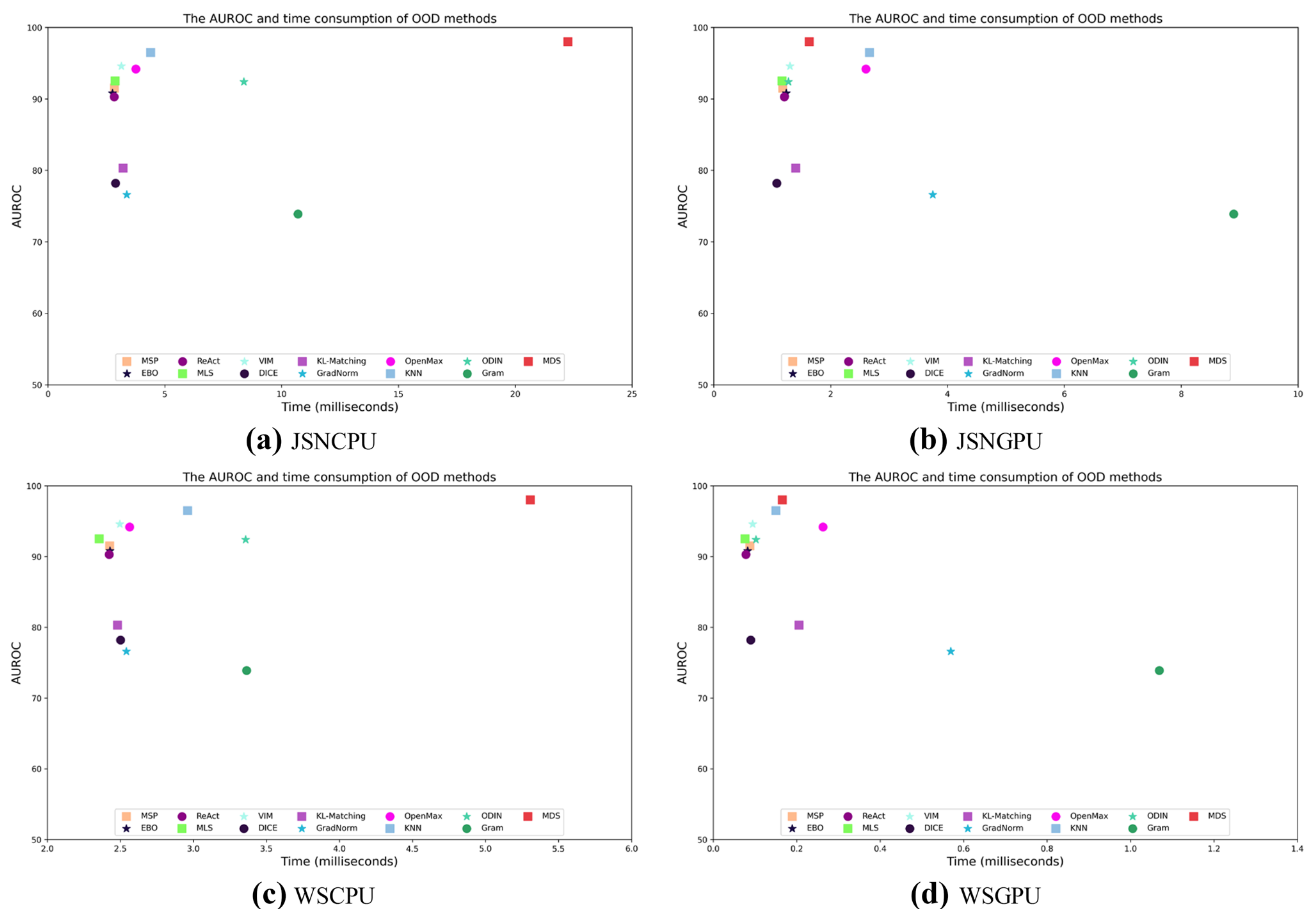
$$M_l = \max_c - (f_l(\hat{x}) - \hat{\mu}_{l,c})^{\mathrm{T}} \hat{\Sigma}_l^{-1} (f_l(\hat{x}) - \hat{\mu}_{l,c}) \quad (11)$$

Finally, return confidence score for the input as $\sum_l \alpha_l M_l$.

One tunable hyperparameter in MDS is the set of layer indices for computing the feature ensemble, as the confidence scores are extracted from every end of residual blocks of ResNet with the specified layer indices. In our experiments, we set $L = 3$, i.e., the set of layer indices $l \in \{1, 2, 3\}$ refer to the first 3 residual blocks in ResNet-18. From Table 7, we can see that the Inf and Backprop steps take much longer than the other algorithms, and Backprop steps take less time than Inf steps since they start from intermediate layers. GPU achieves significant acceleration for both the Inf and Backprop steps, and also the overall algorithm.

4.3.6 KNN

MDS is based on the assumption that samples of each class form a Gaussian distribution in the feature space, which may not always hold true. KNN-based OOD detection (Sun et al. [21]) is a non-parametric distance-based method



(a) JSNCPU **(b)** JSNGPU **(c)** WSCPU **(d)** WSGPU

Figure 5 AUROC vs. execution time with Mnist as the ID dataset, Near-OOD as the OOD dataset.

Table 8 Execution time of different components of KNN (time unit: ms).

KNN	JSNCPU	JSNGPU	WSCPU	WSGPU
Inf	55.65	5.93	5.56	0.15
KNN	9.04	7.48	1.24	0.89

Table 9 Execution time of different components of Gram on the workstation (time unit: ms).

Gram	WSCPU	WSGPU
Inf	4.46	0.15
Gram_M	2.13	0.26 (on CPU)
Deviation	1.74	1.24

that does not impose such distributional assumptions. It is based on the normalized feature vector $z = f(x)/\|f(x)\|_2$ in the penultimate layer. We first collect feature vectors $\mathbb{Z}_n = (z_1, z_2, \ldots, z_n)$ for each input x_i in the training dataset D_{in}. In the testing stage, we first perform forward inference (Inf) to obtain the normalized feature vector z^* from input x^* (Norm), then calculate the Euclidean distances $\|z_i - z^*\|_2$ w.r.t. embedding vectors $z_i \in \mathbb{Z}_n$; reorder $\mathbb{Z}_n$ in the order of increasing distance $\|z_i - z^*\|_2$. Denote the reordered data sequence as $\mathbb{Z}'_n = (z_{(1)}, z_{(2)}, \ldots, z_{(n)})$. OOD detection is based on the following decision function:

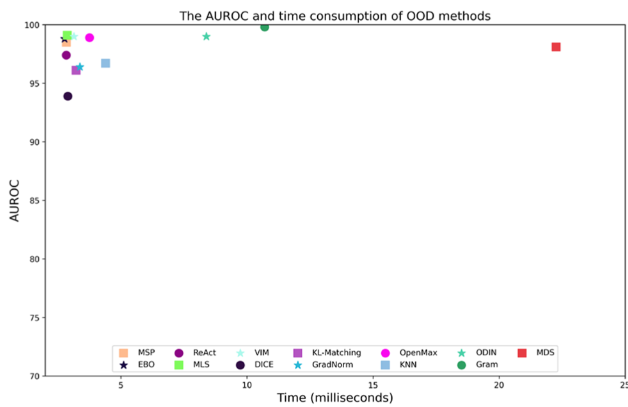
$$G(z^*;k) = 1\left\{-r_k(z^*) \geq \lambda\right\} \tag{12}$$

where $r_k(z^*) = \|z^* - z_{(k)}\|_2$ is the distance to the k-th nearest neighbor, and $1\{\}$ is the indicator function. The threshold λ is chosen so that a large fraction of ID data (e.g., 95%) is correctly classified.

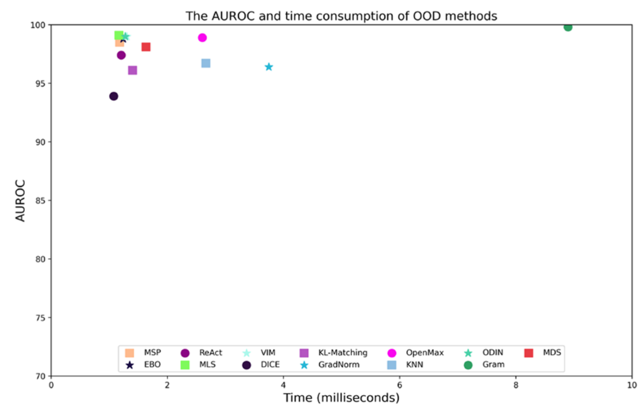
The implementation of KNN relies on the open-source library Faiss [37] for efficient similarity search and clustering of dense vectors. We can specify Faiss to run on either the CPU or the GPU, and the GPU implementation can accept input from either CPU or GPU memory (if they are separate). From Table 8, we can see that GPU achieves a small acceleration for the KNN step on both platforms.

4.3.7 Gram

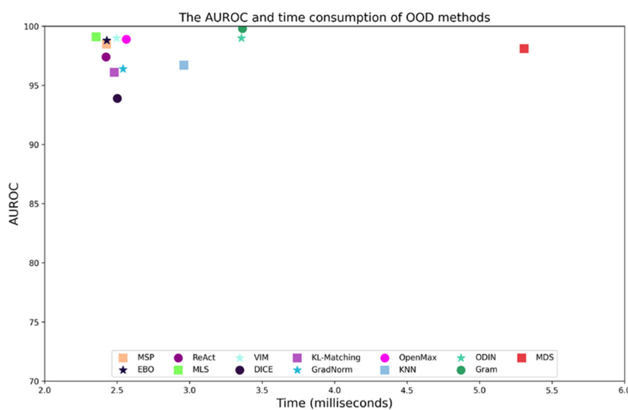
Gram (Sastry et al. [22]) uses higher-order Gram matrices to compute pairwise feature correlations between the channels of each layer of a DNN.



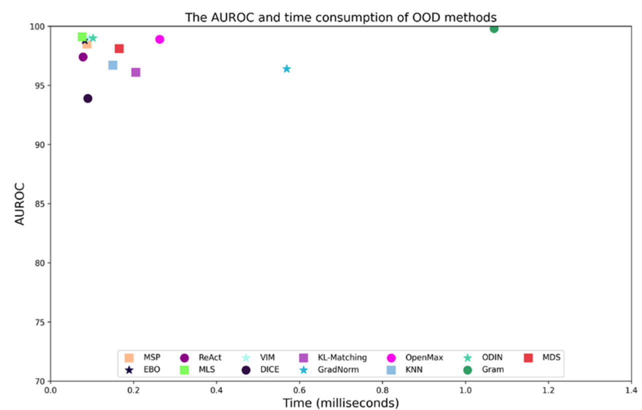
(a) JSNCPU



(b) JSNGPU



(c) WSCPU



(d) WSGPU

Figure 6 AUROC vs. execution time with Mnist as the ID dataset, Far-OOD as the OOD dataset.

We use $F_l(x)$ to denote the feature map at the l-th layer for input image x. It is stored in a matrix of dimension $n_l \times p_l$, where n_l is the number of channels at the l-th layer and p_l, the number of activation values per channel, is the height times the width of the feature map. The p-th order Gram matrix is computed using F_l^p, where the power of F_l is computed element-wise:

$$G_l^p = \left(F_l^p F_l^{pT}\right)^{\frac{1}{p}} \tag{13}$$

We use $\overline{G_l^p}$ to denote the flattened upper (or lower) triangular matrix along with the diagonal entries. We set $p = 5$ in our experiments.

Gram consists of three steps. It first extracts the feature vector $F_l(x)$ from input x; then for each of the layers, computes $\overline{G_l^p}$ (Gram_M) on the CPU, then computes layer-wise deviations (Deviation) of the input from the input images seen at training time, as the percentage change w.r.t the maximum or minimum values of feature co-occurrences, on the GPU.

Table 9 shows the results on the workstation only, since Gram runs out of memory on Jetson Nano. We can see that GPU helps achieve significant acceleration for each of the three steps of Gram.

4.4 AUROC vs. Execution Time

In this section, we plot both the metrics of AUROC and execution time in the same figure to have a more complete picture of the overall performance in terms of both metrics. We state that algorithm A *dominates* algorithm B if A has both *higher AUROC* and *shorter execution time* than B, i.e., A lies on the left and on the top of B in subsequent figures. Therefore, the algorithm on the upper-left corner dominates all the others (if there exists one).

4.4.1 Mnist Near-OOD Benchmark

Figure 5 shows the results when the ID dataset is Mnist, and the OOD datasets are Near-OOD.

From Figure 5(a), (c), we can see that on JSNCPU and WSCPU, MDS lies in the upper right corner, with the highest AUROC and longest execution time. VIM and MLS lie in the upper left corner, and outperform most other algorithms

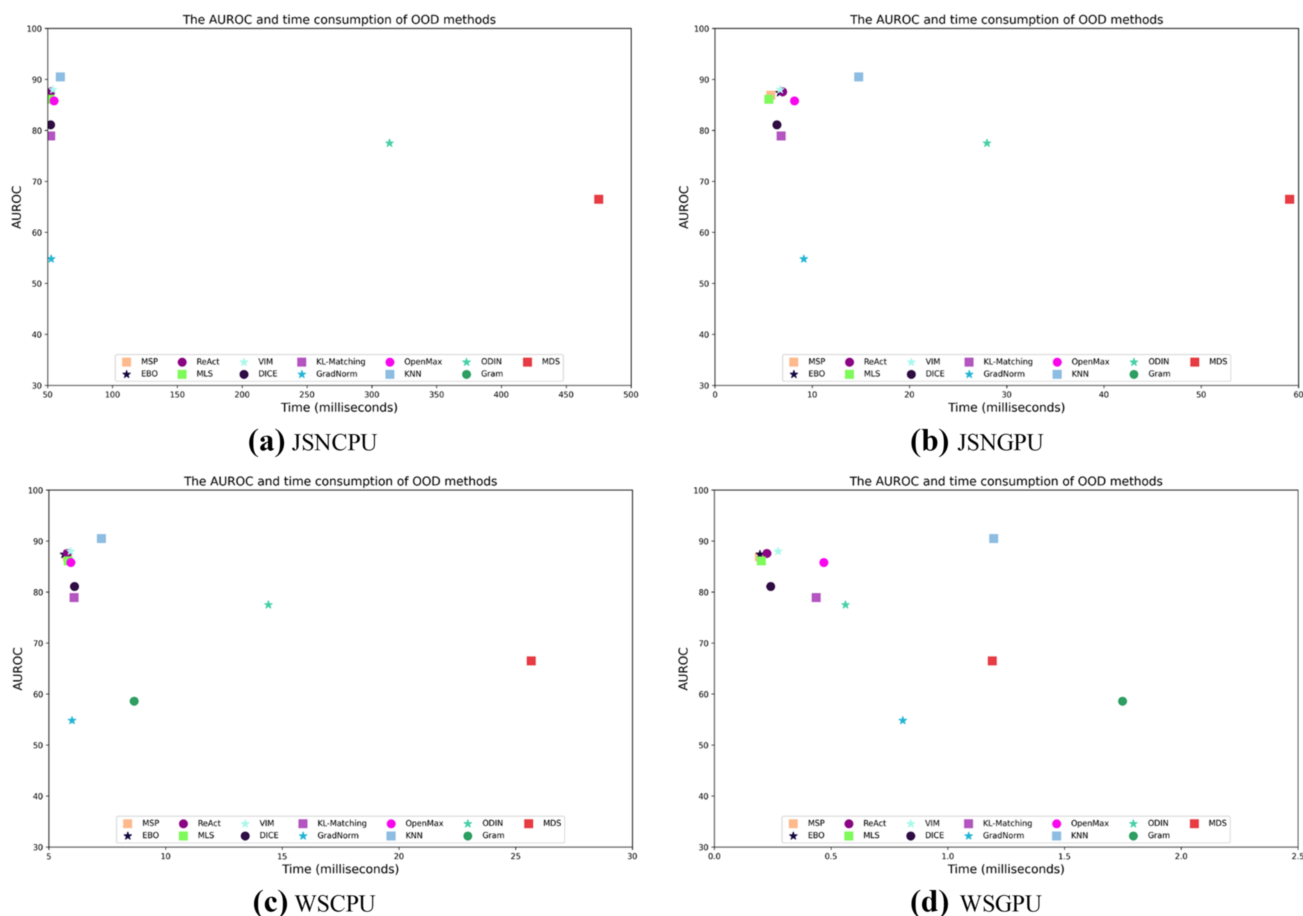


Figure 7 AUROC vs. execution time with Cifar10 as the ID dataset, Near-OOD as the OOD dataset. (Gram is not shown in (a)(b), since it runs out of memory on Jetson Nano.)

in both AUROC and execution time. Gram has relatively low AUROC and medium execution time.

From Figure 5(b), (d), we can see that on JSNGPU and WSGPU, MDS has the highest AUROC and relatively short execution time, thanks to the significant acceleration effect of the GPU. Gram performs poorly with the lowest AUROC and longest execution time.

4.4.2 Mnist Far-OOD Benchmark

Figure 6 shows the results when the ID dataset is Mnist, and the OOD datasets are Far-OOD.

From Figure 6(a), (c), we can see that on JSNCPU and WSCPU, MDS lies close to the upper right corner, with relatively high AUROC and long execution times. Gram has the highest AUROC and medium execution time.

From Figure 6(b), (d), we can see that on JSNGPU and WSGPU, Gram lies in the upper right corner, with the highest AUROC and longest execution time. While Gram is significantly accelerated by GPU, MDS is accelerated even more.

4.4.3 Cifar10 Near-OOD Benchmark

Figure 7 shows the results when the ID dataset is Cifar10, and the OOD datasets are Near-OOD.

From Figure 7(a), (c), we can see that on JSNCPU and WSCPU, MDS has medium AUROC and the longest execution time. From Fig. 7(d), we can see that on WSGPU, Gram is the most time-consuming algorithm.

4.4.4 Cifar10 Far-OOD Benchmark

Figure 8 shows the case when the ID dataset is Cifar10, and the OOD datasets are Far-OOD.

From Fig. 8(a), (c), we can see that on JSNCPU and WSCPU, MDS lies close to the upper right corner, with relatively high AUROC and long execution times. From Fig. 8(d), we can see that on WSGPU, Gram is the most time-consuming algorithm.

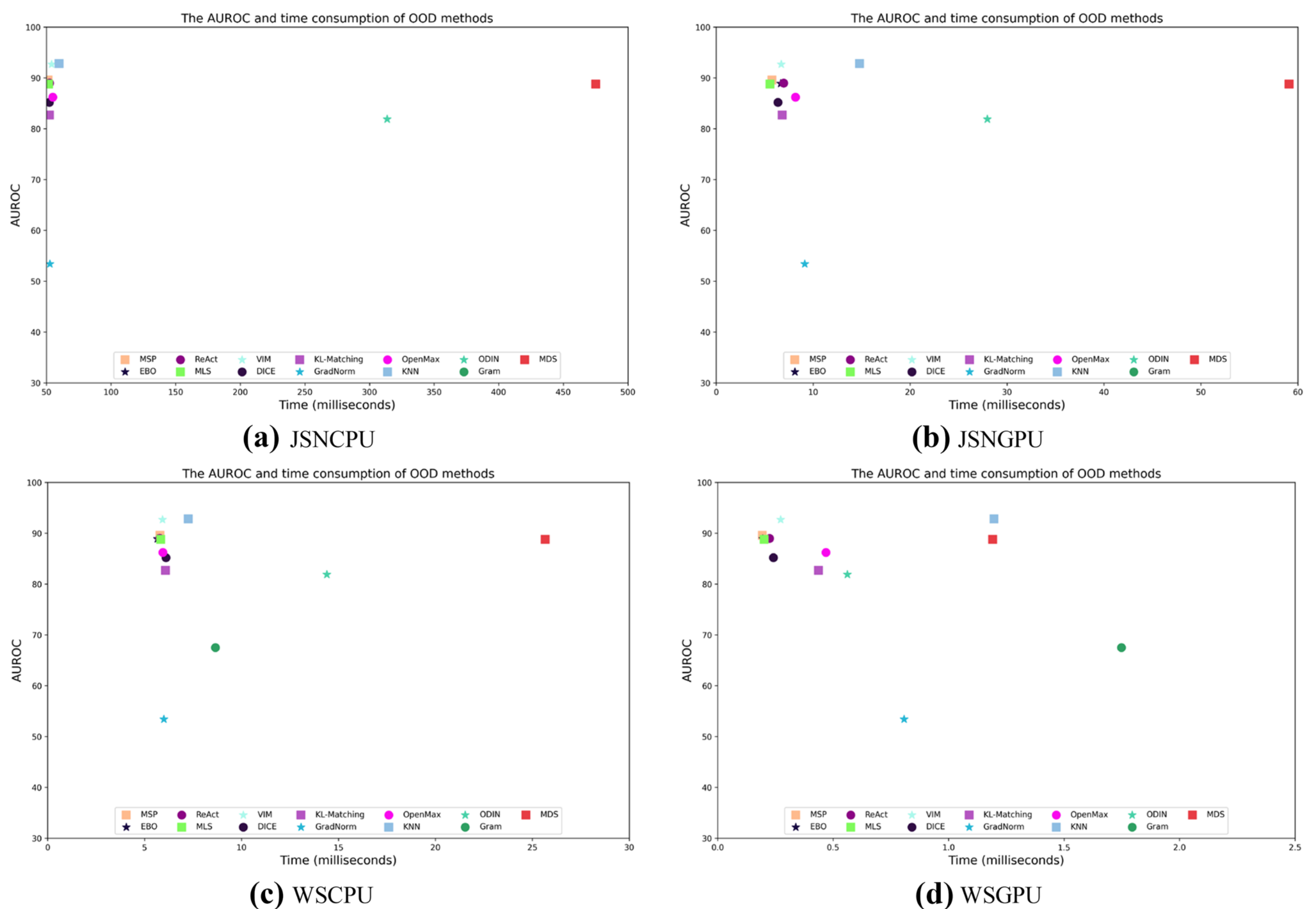


Figure 8 AUROC vs. execution time with Cifar10 as the ID dataset, Far-OOD as the OOD dataset. (Gram is not shown in (a)(b), since it runs out of memory on Jetson Nano.)

4.5 Discussions

We make the following observations from our experimental results:

1. Overall, no OOD detection algorithm consistently dominates all the others, and there is no monotonic correlation between accuracy and execution time, i.e., a more time-consuming algorithm may not necessarily outperform another simple-and-efficient algorithm, and the algorithm accuracy is dependent on multiple factors, incl. DNN architectures and hyperparameters, ID and OOD datasets, etc. This observation is consistent with Tajwar et al. [38], who showed that none of the three OOD detection algorithms (MSP, ODIN, and MDS) is consistently better than the others on a standardized set of 16 (ID, OOD) pairs.
2. While there is no consistent winner, we can observe from Figs. 5, 6, 7, and 8 (AUROC vs. execution time) that several algorithms form a tight cluster in the upper left corner, with relatively high AUROC values and low execution times, including MSP, EBO, MLS, VIM, and React. Hence they can be the safe choice of OOD detection algorithms for most application scenarios. KNN, Gram, and MDS can generally achieve high AUROC, but they have relatively long execution times.
3. GPU helps achieve significant acceleration for the DNN forward inference and back-propagation of gradients, hence OOD detection algorithms such as MDS and ODIN benefit the most from GPU acceleration, since they both incur two DNN forward inferences and one back-propagation of gradients.

5 Conclusions and Future Work

OOD detection is an important topic for the practical deployment of DNNs in safety–critical applications. During the actual deployment, the designer needs to select the most appropriate OOD detection algorithm according to application requirements and available hardware resources. In this paper, we carry out comprehensive and systematic benchmarking and evaluation of both accuracy and execution time metrics of well-known OOD detection algorithms on different hardware platforms, to provide a useful reference for the system designer in choosing the most appropriate OOD detection algorithm for given application requirements and hardware capability.

All the DNN models and OOD detection algorithms studied in this paper are implemented in Python, which is an interpreted language that is much slower than compiled languages such as C/C++. For practical deployment on resource-constrained embedded platforms, it may be necessary to perform model compression [39–41], and compile the DNNs into efficient executable code with a Deep Learning compiler framework [42]. This also requires the OOD score function to be converted from Python to C/C++ to be used in conjunction with the compiled DNN.

Author contributions System implementation and evaluation (S. Luan); Idea conception and paper writing (Z. Gu); Method formulation (A. Saremi, L. Freidovich, L. Jiang, S. Wan).

Funding Open access funding provided by Umea University. This work was partially supported by National Natural Science Foundation of China under Grant No. 62172438, Key Project of Shenzhen City Special Fund for Fundamental Research under Grant # 202208183000751, and the Kempe Foundation, Sweden.

Data Availability N/A.

Declarations

Ethics Approval This research involves no human participants and/or animals.

Competing Interests The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Yang, J., Zhou, K., Li, Y., & Liu, Z. (2021). Generalized out-of-distribution detection: A survey. arXiv preprint: arXiv:2110.11334
2. Yang J., et al. (2022) OpenOOD: Benchmarking Generalized Out-of-Distribution Detection. arXiv preprint: arXiv:2210.07242
3. Shen Z., et al. (2021). Towards out-of-distribution generalization: a survey. arXiv preprint: arXiv:2108.13624
4. Gu, Z., Wang, S., Kodase, S., & Shin, K. G. (2003). An End-to-End Tool Chain for Multi-View Modeling and Analysis of Avionics Mission Computing Software. In RTSS 2003. 24th IEEE Real-Time Systems Symposium: IEEE Computer Society, pp. 78–78.
5. Al-bayati, Z., Zhao, Q., Youssef, A., Zeng, H., & Gu, Z. (2015) Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms. In the 20th Asia and South Pacific Design Automation Conference: IEEE pp. 630–635.
6. Xia, X., et al. (2022). GAN-based anomaly detection: A review. *Neurocomputing, 493*, 497–535.
7. Cai, F., Ozdagli, A. I., & Koutsoukos, X. (2022). Variational Autoencoder for Classification and Regression for Out-of-Distribution

Detection in Learning-Enabled Cyber-Physical Systems. *Applied Artificial Intelligence, 36*(1), 2131056.

8. Henzinger, T. A., Lukina, A., & Schilling, C. (2019). Outside the box: Abstraction-based monitoring of neural networks. arXiv preprint: arXiv:1911.09032
9. Luan, S., Gu, Z., Freidovich, L. B., Jiang, L., & Zhao, Q. (2021). Out-of-distribution detection for deep neural networks with isolation forest and local outlier factor. *IEEE Access, 9*, 132980–132989.
10. Hendrycks, D., & Gimpel, K. (2016). A baseline for detecting misclassified and out-of-distribution examples in neural networks. arXiv preprint: arXiv:1610.02136
11. Zhao, Q., Chen, M., Gu, Z., Luan, S., Zeng, H., & Chakrabory, S. (2022). CAN bus intrusion detection based on auxiliary classifier GAN and out-of-distribution detection. *ACM Transactions on Embedded Computing Systems (TECS), 21*(4), 1–30.
12. Liang, S., Li, Y., & Srikant, R. (2017). Enhancing the reliability of out-of-distribution image detection in neural networks. arXiv preprint: arXiv:1706.02690
13. Hendrycks, D., et al. (2022). Scaling out-of-distribution detection for real-world settings. In International Conference on Machine Learning, PMLR, pp. 8759–8773.
14. Wang, H., Li, Z., Feng, L., & Zhang, W. (2022). ViM: Out-Of-Distribution with Virtual-logit Matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4921–4930.
15. Huang, R., Geng, A., & Li, Y. (2021). On the importance of gradients for detecting distributional shifts in the wild. *Advances in Neural Information Processing Systems, 34*, 677–689.
16. Sun, Y., & Li, Y. (2022). Dice: Leveraging sparsification for out-of-distribution detection. *European Conference on Computer Vision* (pp. 691–708). Springer.
17. Sun, Y., Guo, C., & Li, Y. (2021). React: Out-of-distribution detection with rectified activations. *Advances in Neural Information Processing Systems, 34*, 144–157.
18. Liu, W., Wang, X., Owens, J., & Li, Y. (2020). Energy-based out-of-distribution detection. *Advances in neural information processing systems, 33*, 21464–21475.
19. Bendale, A., & Boult, T. E. (2016). Towards open set deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1563–1572.
20. Lee, K., Lee, K., Lee, H., & Shin, J. (2018). A simple unified framework for detecting out-of-distribution samples and adversarial attacks. arXiv preprint: arXiv:1807.03888
21. Sun, Y., Ming, Y., Zhu, X., & Li, Y. (2022). Out-of-distribution detection with deep nearest neighbors. In *International Conference on Machine Learning*, PMLR, pp. 20827–20840.
22. Sastry, C. S., & Oore, S (2020) Detecting out-of-distribution examples with gram matrices. In *International Conference on Machine Learning*, PMLR, pp. 8491–8501.
23. Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine, 29*(6), 141–142.
24. Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical Report, University of Toronto.
25. Bulatov, Y. NotMnist dataset. Retrieved March 1, 2023, from http://yaroslavvb.com/upload/notMNIST
26. Xiao, H., Rasul, K., & Vollgraf, R. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint: arXiv:1708.07747
27. Kylberg, G. (2011). Kylberg texture dataset v. 1.0. Centre for Image Analysis, Swedish University of Agricultural Sciences and Uppsala University.
28. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM, 60*(6), 84–90.
29. Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., & Torralba, A. (2017). Places: A 10 million image database for scene recognition. *IEEE transactions on pattern analysis and machine intelligence, 40*(6), 1452–1464.
30. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.
31. Krizhevsky, A., Nair, V., & Hinton, G. Cifar-10 and cifar-100 datasets. Retrieved March 1, 2023, from https://www.cs.toronto.edu/kriz/cifar.html
32. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., & Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning.
33. He, K., Zhang, X., Ren, S., & Sun, J. (2016) Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778.
34. Harris, C. R., et al. (2020). Array programming with NumPy. *Nature, 585*(7825), 357–362.
35. Pedregosa F., et al. (2011) Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830.
36. Tripathy, A. Timing your PyTorch Code Fragments. Retrieved March 1, 2023, from https://auro-227.medium.com/timing-your-pytorch-code-fragments-e1a556e81f2
37. Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data, 7*(3), 535–547.
38. Tajwar, F., Kumar, A., Xie, S. M. & Liang, P. (2021). No true state-of-the-art? OOD detection methods are inconsistent across datasets. arXiv preprint: arXiv:2109.05554
39. Choudhary, T., Mishra, V., Goswami, A., & Sarangapani, J. (2020). A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, pp. 1–43.
40. Luan, S., Gu, Z., Xu, R., Zhao, Q., & Chen, G. (2023) LRP-based network pruning and policy distillation of robust and non-robust DRL agents for embedded systems. *Concurrency and Computation: Practice and Experience*.
41. Meng, W., Gu, Z., Zhang, M., & Wu, Z. (2017). Two-bit networks for deep learning on resource-constrained embedded devices. arXiv preprint: arXiv:1701.00485
42. Li, M., et al. (2020). The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems, 32*(3), 708–727.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.