SysNoise: Exploring and Benchmarking Training-Deployment System Inconsistency

Anonymous Authors[1]

Abstract

Extensive studies have shown that deep learning models are vulnerable to adversarial and natural noises, yet little is known about model robustness on noises caused by different system implementations. In this paper, we for the first time introduce SysNoise, a frequently occurred but often overlooked noise in the deep learning training-deployment cycle. In particular, SysNoise happens when the source training system switches to a disparate target system in deployments, where various tiny system mismatch adds up to a non-negligible difference. We first identify and classify SysNoise into three categories based on the inference stage; we then build a holistic benchmark to quantitatively measure the impact of SysNoise on 20+ models, comprehending image classification, object detection, instance segmentation and natural language processing tasks. Our extensive experiments revealed that SysNoise could bring certain impacts on model robustness across different tasks and common mitigations like data augmentation and adversarial training show limited effects on it. Together, our findings open a new research topic and we hope this work will raise research attention to deep learning deployment systems accounting for model performance. We have open-sourced the benchmark and framework at `https://modeltc.github.io/systemnoise_web`.

1 Introduction

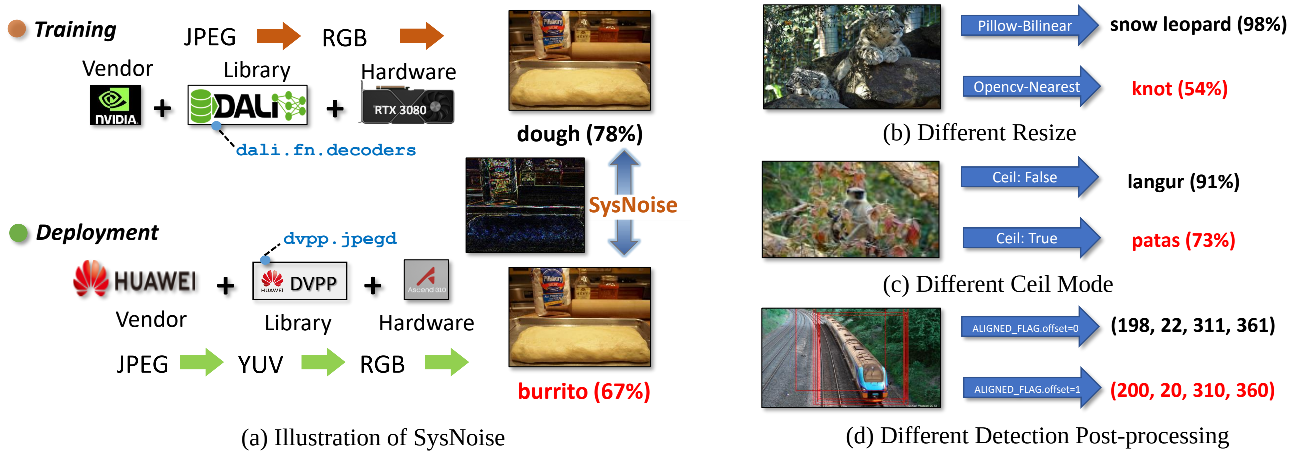
Deep neural networks have demonstrated remarkable success in handling multiple tasks (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014; He et al., 2016a; Devlin et al., 2018; Brown et al., 2020), yet they are vulnerable against noises. Despite the progress devoted to noises made by human-being or nature (*e.g.*, adversarial noises (Goodfellow et al., 2014b) and natural noises (Hendrycks & Dietterich, 2019)), little is known about model robustness on noises caused by different system implementations. In practice, the model deployment often faces diverse implementation platforms spanning from general (*e.g.*, CPU, GPU) to specialized (*e.g.*, NPU, ASIC) computing hardware; from the cloud server to edge devices; and often with different backends (*e.g.*, TensorRT (NVIDIA) for GPUs, SNPE (Qualcomm) for DSPs, CANN (HUAWEI) for Ascend). These different software-hardware system implementations would bring certain noises resulting in considerable model performance degeneration. More importantly, these noises cannot be completely prohibited as long as a trained model will be deployed to multiple target platforms.

Thus, in this paper, we pioneeringly discuss an unwanted yet non-negligible type of noise caused by the inconsistency of the training-deployment system (see Fig. 1 for illustration), deemed as system noise (*abbrev. SysNoise*). Based on where SysNoise could happen, we classify it into three different types. ① *Pre-processing:* Depends on the implementation of input data. For example, different image decoding (JPEG2RGB) algorithms and different interpolation methods for image resize and crop. ② *Model Inference:* Caused by different implementations of the model during inference. For instance, models with the same parameters can have different results when the upsampling operator is different. Using different data types (INT8, FP16, FP32) also leads to different accuracy. ③ *Post-processing:* Includes the further manipulation of inference results, *e.g.*, applying softmax function in classification tasks and calculating the bounding box in detection tasks. Overall, SysNoise exhibits its impact on the whole inference pipeline, leading to an undesired performance drop.

To better understand and comprehensively evaluate the influence of SysNoise on the deployed model, we provide a thorough quantitative benchmark on 3 common computer vision tasks (*i.e.*, classification, detection, and segmentation) with 20+ representative models and typical baselines. As for natural language processing, we provide a benchmark on OPT (Zhang et al., 2022) model on 4 datasets. Our large-scale experiments reveal several insights: (1) though these noises are not chosen by any adversary, SysNoise would bring considerable impacts on model robustness, and could cause up to 9.97% and 10.67% drops on classification and detection tasks respectively; (2) different architecture fami-

[1] Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.



(a) Illustration of SysNoise

(b) Different Resize

(c) Different Ceil Mode

(d) Different Detection Post-processing

Figure 1: An illustration of SysNoise (a) and its negative effect on model robustness (b-d). Here we take noises from the decoder as an example. We usually use the DALI library from NVIDIA on GPU during training and the DVPP library from HUAWEI on Ascend during deployment for decoding acceleration, which results in minor decoding differences and would mislead the prediction.

lies induce different robustness on SysNoises (*e.g.*, ViTs and CNNs), even in the same architecture family, a larger model tends to have low variance and low accuracy degradation on SysNoise; and (3) SysNoise seems to be highly diverse and different from adversarial and natural noises, where common mitigations like data augmentation and adversarial training show limited effects on it. Together with existing benchmarks on adversarial and natural noises, we could build a more comprehensive and general understanding and ecosystems for robustness benchmarking involving more perspectives. This benchmark for evaluating robustness to system noises provides useful information, and hopefully, it can open a new research direction for building robust deep learning deployment systems.

In conclusion, our contributions can be summarized as three-fold:

1. For the first time, we identify an important yet long-neglected problem named SysNoise (ranging from pre-processing, model inference, and post-processing noise), which is caused by the training-deployments system inconsistency.
2. We build a benchmark and framework to quantitatively evaluate SysNoise on 20+ deep neural networks, including image classification (ImageNet), detection (MS COCO), segmentation (CitySpace), and natural language processing.
3. We conducted in-depth analyses and found several insights, which revealed that SysNoise is an inevitable and urgent-to-solve problem for both algorithm researchers and hardware vendors.

2 Related Work

Noises Types and Benchmarks. Extensive shreds of evidence have shown that deep learning models are unstable towards different noises, including adversarial noises and natural noises. *Adversarial noises*, which are imperceptible to human vision, could easily make neural networks misclassify the input images (Szegedy et al., 2013; Goodfellow et al., 2014a; Madry et al., 2018; Carlini & Wagner, 2017; Liu et al., 2019; Liang et al., 2021; Wang et al., 2021). To benchmark and evaluate adversarial robustness, (Su et al., 2018) first investigated the adversarial robustness of 18 models on ImageNet; (Xiang et al., 2019) built the platform DEEPSEC for adversarial robustness analysis including 16 adversarial attacks, and 13 adversarial defenses; meanwhile, RealSafe (Yinpeng et al., 2020) open-sourced and benchmarked adversarial robustness on image classification tasks. More recently, large-scale benchmarks on adversarial robustness regarding defense strategies (RobustBench (Croce et al., 2020)) and model architectures (RobustART (Tang et al., 2021)) were developed. Besides adversarial noises, there exist another type of model-agnostic noise named *natural noises* (also deemed as corruptions), which are commonly witnessed in the real-world scenario, *e.g.*, blur, snow, and frost. Some representative datasets are constructed to simulate and benchmark the natural noises, such as ImageNet-P, ImageNet-C (Hendrycks & Dietterich, 2019), and ImageNet-A, ImageNet-O (Hendrycks et al., 2021b). (Hendrycks et al., 2021a) also introduced new real-world distribution shift datasets including changes in image style, geographic location *etc*. However, these studies only focus on noises brought during data acquisition, while ignoring the impacts of the whole inference pipeline caused by different system implementations.

In addition, (Jia & Rinard, 2021) takes the first step towards the influence of the floating-point value representation. They highlight that, to achieve practically reliable verification of neural networks, the system must accurately model the effects of any floating-point computations. However, this paper only conducts a preliminary attempt at the effect of floating-point numerical error for neural network verifiers.

By contrast, *this paper for the first time proposes system noises, which is caused by the training-deployment system inconsistency, and accordingly builds a benchmark to comprehensively study the effect on model performance.*

Approaches to Improving Model Robustness. To improve model robustness against *adversarial noises*, a long line of adversarial defense works have been proposed including: (1) adversarial training that adversarially train deep models using adversarial examples (Goodfellow et al., 2014b; Madry et al., 2018; Tramèr et al., 2017; Shafahi et al., 2019; Liu et al., 2021a); and (2) adversarial detection that distinguishes the clean example and adversarial example (Grosse et al., 2017; Gong et al., 2017; Jiang et al., 2020). To effectively tackle the *natural noises*, several studies have been devoted primarily from the perspective of data augmentation. By producing an elementwise convex combination of two images, Mixup (Zhang et al., 2017) could regularize neural networks to favor simple linear behavior in-between training examples and improve model performance. Different from Mixup, AutoAugment (Cubuk et al., 2018) adopts and tunes a group of augmentations to optimize performance on a downstream task. To further improve model robustness against natural noises, AugMix (Hendrycks et al., 2020) was proposed to mix multiple augmented images. And APR-SP (Chen et al., 2021) was proposed to force the CNN to pay more attention on the structured information from phase components and keep robust to the variation of the amplitude which can help with the model's robustness of natural noise.

3 System Noise Benchmark

In this section, we introduce the benchmark for system noise. First, we summarize the three stages in SysNoise, namely pre-processing noise, model inference noise, and post-processing noise as shown in Fig. 2. Then, we introduce these three stages SysNoise in detail. Note that we only give the basic principles, a more rigorous mathematical difference of SysNoise is provided in Appendix A.

3.1 Pre-processing Noise

Pre-processing means the preparation of the input tensor of the neural network. Concretely, in computer vision tasks, the pre-processing will convert an image raw file to a 3-dimension tensor (width, height, and RGB channels). To fulfill this conversion, two steps are required. First, the raw file (JEPG) will be decoded to a tensor with the image's original shape. Then, the tensor will be resized to a certain shape. To decode the image from the JEPG file to an RGB tensor, it is required to perform the inverse Discrete Cosine Transform (iDCT) operation. In theory, the principle of iDCT is fixed, but we find decoding one image file in different third-party libraries (*e.g.*, OpenCV (Bradski, 2000), Pillow (Umesh, 2012), FFmpeg (Tomar, 2006b)) will output different RGB tensors. This is because some libraries prefer to use Fast iDCT (Chen et al., 1977) instead of the vanilla one, which may sacrifice the image quality for the decoding speed. Furthermore, there would be some minor errors in the decoding implementation, such as the cosine function. These minor errors can cause a shift in the pixel values of the final RGB image tensor. As a result, when changing the decoding tools used in training to another one in inference, we observe a drop in accuracy.

The second cause of pre-processing noise is image resize. Resize is a simple scaling operation that adjusts resolution to a different size, either up (increase resolution) or down (decrease resolution). In a resize operation, one needs to predict the pixel value at an unseen position. This is often performed by different *interpolation* algorithms. For example, nearest-neighbor interpolation directly selects the value of its nearest know pixel. While bilinear interpolation predicts the unknown pixel by computing the distance-based weight average of the existing neighbor 4 pixels, *i.e.* top, bottom, left, and right, which has a rather continuous interpolation effect. Besides, there are many other interpolation methods. In Appendix, we provide the detailed mathematical explanation of these interpolation algorithms as well as the supporting package in computer vision. Note that the difference in interpolation may even occur at the package level, *i.e.* even the same interpolation algorithm might differ in different supporting packages.

The third source of inconsistency during the pre-processing stage comes from the conversion of color space. In the practical application, there are various representation formats for videos and images, *e.g.*, RGB and YUV. The RGB format defines the color space with the value of red, green, and blue channels while the YUV format separates the brightness information (Y) from the color information (U and V), which is the format native to TV broadcast and composite video signals. To save the required storage, different variants of the YUV format are devised. Among them, the NV12 format can encode one pixel with only 12bits, enjoying a low memory consumption and high efficiency. Therefore, many decoder accelerators such as Microsoft DirectX Video Acceleration and Ascend 310 adopt this format. However, for the training of most neural networks, input images are fed with the RGB format. Decoding images to YUV and then

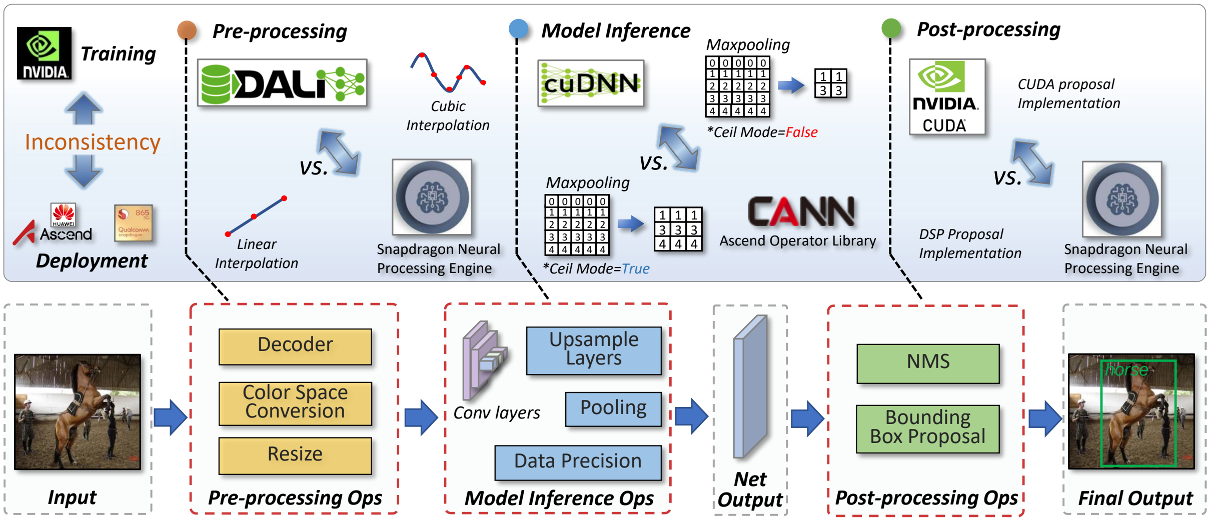


Figure 2: **Overview of SysNoise**. SysNoise is caused by an inconsistent implementation between the training system and deployment system, consisting of three parts, namely pre-processing noise, model inference noise, and post-processing noise.

converting it to RGB is difficult to output the same direct RGB decoded images.

3.2 Model Inference Noise

Model inference noise accounts for the difference that happens during the inference process. This is primarily due to the implementation of various operations. For example, the convolution can be implemented in many ways (GEMM, Img2Col, Winograd, etc). We primarily discover 3 three types of model inference noise that cause a performance drop. The first one is the ceiling mode for max-pooling layers. Ceiling mode means how to compute the output spatial shape. Setting ceiling mode to true will allow the sliding windows to go off-bounds if they start within the left padding. Hardware vendors usually support different ceiling modes, causing an inevitable mismatch.

Another important type of model inference noise is the upsampling method. It is widely used in segmentation task. And in the detection task, the widely used feature pyramid networks (Lin et al., 2017a) integrate the features from different stages within the network. These features have an uneven resolution, requiring an upsampling operation to match feature resolution. Same as the resize operation we discussed in Sec. 3.1, the choice of the interpolation in upsampling layers can play an important role and lead to different predictions. We find that the FPN is quite sensitive to interpolation.

Finally, the precision of data representation can also be viewed as a type of model inference noise. Generally, the input data and the parameters in the model are stored with 32-bit floating-point numbers. However, some hardware systems may restrict the precision, *e.g.*, only 16-bit floating-point numbers or 8-bit integers are allowed. Low-bit numbers unavoidably preserve less information than the full-precision numbers, causing accuracy degradation. Note that in the field of quantization research, some training methods could alleviate this problem (Jacob et al., 2018). We do not use such a training-compensated method here, in order to evaluate how much the deep learning model can resist under low data precision and how a single type interacts with other types of SysNoise, even though there are contingency methods.

3.3 Post-Processing Noise

Post-processing is used to convert the network output to the prediction results. In image classification, this refers to the $\mathrm{Softmax}$ function which applies the exponential function to normalize the output to $(0, 1)$. In object detection, the predicted output of the network needs to be calculated to the final bounding box. During this process, there are rounding operations to get integer resolution coordinates. Then, all the candidate bounding boxes will be sorted with the predicted confidence and filtered with non-maximum suppression. This procedure is easy to introduce noises in detail, *e.g.*, the rounding up or rounding down choice, *etc*. Many hardware vendors provide black-box implementations of these operations to accelerate the deployment. Unfortunately, we find that they often fail to produce the same results, causing an impact on the final performance.

3.4 Benchmarking SysNoise

Types of SysNoise. SysNoise originates from the implementation difference in hardware and software. In Table 1, we briefly summarize the types of SysNoise in each stage, as well as their applied task, dependence on input data, level of effect, and the number of categories. We highlight that here we view SysNoise as random noise since in practice it is inflexible to train a unique model for corresponding hardware.

- *Preprocessing Noise*

1. Decoder: To simulate noise during decoding process, four different python packages are selected to decode images — PIL (Umesh, 2012), OpenCV (Bradski, 2000), FFmpeg (Tomar, 2006a) and DALI (Nvidia), which implement their own image decode function, and output different image tensors.
2. Resize: We choose up to 11 different resize methods to represent noise that occurred in image resizing. Specifically, we utilize two Python packages, the Pillow and the OpenCV. For Pillow, we adopt interpolations from {bilinear, nearest, box, hamming, bicubic, lanczos} methods, and for OpenCV, we adopt interpolations from {bilinear, nearest, area, bicubic, lanczos}.
3. Color mode: To simulate noise that comes from the conversion of color space, we generate the noised images by first decoding the images to RGB and then transforming them to YUV color space and then back to RGB with Ascend Computing Language (ACL) (HUAWEI).

- *Model Inference Noise*

1. Ceil mode: This can only be tested on models which has stride 2 max-pooling layers, such as ResNets (He et al., 2016b). We train the model with floor mode but test it with ceil mode.
2. Upsample: Nearest neighbor and bilinear are the two most commonly supported algorithms for upsampling. Following (Lin et al., 2017b), we train the original upsample layers with nearest-neighbor interpolation and test it with bilinear interpolations.
3. Data Precision: To evaluate the model's robustness under different precisions, we quantize the model to FP16 or INT8 and test it.

- *Postprocessing Noise*

1. Detection proposal: We evaluate the influence of whether to add the value of 1 when calculating bounding boxes from offsets, both of which are common in hardware implementations.

Evaluation Metrics. For classification/detection/segmentation/natural language processing, we report the top-1 accuracy/mean Average Precision/mean Intersection over Union difference for measuring the robustness of models. If the SysNoise has multiple options, we report the mean difference as well as the max difference, otherwise, only the metric difference is reported.

4 Experiment and Analysis

In this section, we conduct a thorough benchmark and analysis on the SysNoise. In Sec. 4.1, we illustrate the experimental setting for image classification, detection, segmentation, and NLP tasks; in Sec. 4.2 we extensively evaluate all types of SysNoise on these four tasks; in Sec. 4.3, we interpret the SysNoise by comparing it with natural noise and adversarial noise as well as some visualizations.

4.1 Experimental Setting

Classification Task. We benchmark SysNoise on the ImageNet dataset for the classification task, including both Convolutional Neural Networks (CNNs) and Vision Transformers (ViTs). For CNNs, we evaluate ResNet (He et al., 2016b), MobileNetV2 (Sandler et al., 2018), RegNet (Radosavovic et al., 2020), and EfficientNet (Tan & Le, 2020) families. In addition, we evaluate an extremely small architecture — MCUNet (Lin et al., 2020), which only has 0.74MB parameters. For ViTs we evaluate the original Vision Transformer (Dosovitskiy et al., 2021) and the Swin Transformer (Liu et al., 2021b) families. Each family covers different computation and memory budgets to ensure both large and tiny models are verified. During training, we use Nvidia DALI (NVIDIA) to prepare data, *i.e.*, image decode, resize and color space are configured by default function in DALI. All models take an input shape of 224×224 except EfficientNet. We train the default model using FP32 format as this is the standard format in GPU training. For ResNet, we train it with the floor mode of its max-pooling layer. All other training settings follow the original settings of the model.

Detection and Segmentation Task. For object detection, we use COCO dataset and adopt 3 backbones: ResNet-34, ResNet-50, and MobileNetV2 in both Faster RCNN (Ren et al., 2015) with FPN (Lin et al., 2017b) and RetinaNet (Lin et al., 2017c). We use the CitySpace dataset to benchmark SysNoise on Segmentation Task, where we evaluate two architectures (Deeplabv3 and U-Net). As for DeepLabv3, following (Chen et al., 2017), Resnet-50 and Resnet-101 backbones are used. During Training, we use the Pillow package and choose bilinear as an image resize interpolation method to prepare data. Following (Lin et al., 2017b) ,we resize images by keeping the ratio the same as the original image and make the maximum size of the image to be 1333×800. Following common practice, all backbones are pre-trained on ImageNet. We train the default model using FP32 format and train the original upsample layers

Table 1: **List of our discerned system noise,** including 3 stages (pre-processing, model inference, post-processing). Affected tasks consists of image classification (Cls), object detection (Det), semantic segmentation (Seg)

and natural language processing(NLP)

Stage	**Pre-processing**			**Model inference**			**Post-processing**
Type	Decoder	Resize	Color Space	Ceil Mode	Upsample	Data Prec.	Detection Proposal
Task	Cls/Det/Seg	Cls/Det/Seg	Cls/Det/Seg	Cls/Det/Seg	Det/Seg	Cls/Det/Seg/NLP	Det
Input Dependence	✗	✗	✓	✗	✗	✓	✗
Noise Effect Level	High	Very High	Middle	High	Very High	High	Middle
Number of Categories	4	11	2	2	2	3	2

with the nearest-neighbor interpolation. For the models with the ResNet backbone, we train it with the floor mode of its max-pooling layer. All other implementations follow the original settings of the model.

Natural Language Processing Task For natural language processing tasks, we use pre-trained OPT (Zhang et al., 2022) models which are transformer-based models with 125M to 175B parameters. For different natural language processing tasks, we use different datasets including PIQA(Bisk et al., 2020), LAMBADA (Paperno et al., 2016), HellaSwag (Zellers et al., 2019) and WINOGRANDE (Sakaguchi et al., 2019). Compared with computer vision, natural language tasks have less noise during pre-processing and post-processing progress. For simplicity, we use model inference noise, or data precision noise to measure SysNoise in these tasks.

To benchmark the robustness against SysNoise, we train deep neural networks with one fixed setting, also commonly used in the PyTorch framework, and evaluate the task performance under other settings depending on the different types of SysNoise (Sec. 3.4).

4.2 Experimental Results

Impact from single SysNoise. Our evaluation is summarized in Table 2 for ImageNet classification, Table 3 for COCO detection, Table 4 for CitySpace segmentation, and Table 5 for natural language processing. It can be observed that different types of SysNoise cause different levels of performance drop. *For classification*, The color mode and FP16 precision have a subtle impact on the performance of CNNs, while the image decode and resize can have a 0.6-2.3% accuracy decrease on average. In model inference noise, the ceiling mode has a profound effect, where the accuracy drops by 0.8-2.7%. *For detection and segmentation tasks*, there are extra types of SysNoise, the interpolation for upsample layer, and the proposal operation for post-processing. Notably, these two types cause a considerable performance drop. They cause Faster RCNN with ResNet-50 backbone drop of 1.7 and 2.4 mAP, respectively. *For natural language processing tasks*, the impact of data precision has a greater relationship with the datasets. In addition, we find SysNoise behaves differently at the task level. For example, The resize noise has a relatively larger impact on the detection task than the classification task. While the decoder noise nearly has no impact on detection and segmentation tasks but can affect classification models.

Architecture-wise robustness again SysNoise. We also observe some relationships between architecture and SysNoise for classification. *First, in the same architecture family, a larger model tends to have low accuracy degradation.* For instance, in ResNet and RegNet family the average accuracy decrease by decode noise reduces from 1.6% to 0.6% when switching from tiny to large models. The same trend is also found in other noises and tasks. *Second, the lightweight architecture family is more prone to SysNoise.* Specifically, the MobileNetV2 family shows a larger accuracy decrease than other architecture families. The largest MobileNetV2 drops 1.65% accuracy due to different resize methods while the similar-accuracy-level ResNet-50 only drops 0.75%. Furthermore, the MCUNet for STM32F746 with just 320KB memory has the worst robustness among all models, which suffers from an average 4.0% accuracy drop and a maximum 9.3% accuracy drop in resizing noise. *Third, ViTs demonstrate different robustness compared with CNNs.* The Swin Transformers are more robust than CNNs when attacked by decoder noise. Interestingly, both ViTs and Swin Transformers suffer from higher accuracy lost in color mode noise than CNNs. These results demonstrate the extremely high diversity of SysNoise.

Impact from multiple SysNoise. The single noise type may only have limited impacts on task performance. However, it is likely that SysNoise will happen in multiple stages during inference, and will have a combined effect with multiple noises to bring further influences on the accuracy.

We show how combined SysNoise affects a single model step by step in Fig. 3. For example, on ResNet-50, we select the most influential SysNoise type and gradually add them to impose noise coherently. As shown in Fig. 3, we show that some SysNoise is lessened while others are strengthened when combined together. For instance, adding resize to ResNet-50 incurs 0.71% extra accuracy loss which is even lower compared to the average 0.75% accuracy loss in

Table 2: **Measuring SysNoise on ImageNet classification benchmark.** We record Top-1 accuracy and the difference, ΔACC = ACC$_{\text{original}}$-ACC$_{\text{SysNoise}}$. We report both mean and max ΔACC for decode and resize. *The lower* ΔACC *the better.*

Architecture	**Trained**	**Decode**	**Resize**	**Color Mode**	**Precision (FP16/INT8)**		**Ceil Mode**	**Combined**
	ACC	ΔACC	ΔACC	ΔACC	ΔACC	ΔACC	ΔACC	ΔACC
MCUNet-293KB	63.40	0.41 (0.42)	4.02 (9.31)	0.20	0.01	0.04	-	9.97
ResNet18x0.25	48.96	1.98 (2.12)	2.11 (3.71)	0.14	-0.01	0.82	2.34	6.61
ResNet18x0.5	61.64	1.67 (1.76)	1.76 (3.25)	0.19	-0.01	0.15	2.72	6.10
ResNet-18	69.96	1.02 (1.03)	1.01 (2.05)	0.13	0.00	0.20	2.40	4.97
ResNet-34	73.59	0.99 (1.00)	0.77 (1.67)	0.14	0.00	0.04	0.85	4.25
ResNet-50	76.39	0.98 (0.98)	0.75 (1.69)	0.09	0.00	0.06	1.24	3.95
ResNet-101	78.10	0.68 (0.69)	0.62 (1.47)	0.24	0.01	0.69	0.75	4.50
MobileNetV2-0.5	64.94	1.98 (2.00)	2.04 (3.14)	0.18	0.01	0.57	-	5.81
MobileNetV2-0.75	70.26	1.39 (1.39)	1.47 (2.56)	0.16	0.01	0.72	-	5.58
MobileNetV2-1	73.12	1.39 (1.39)	1.48 (2.43)	0.07	0.02	0.77	-	5.03
MobileNetV2-1.4	75.84	1.01 (1.02)	1.65 (2.15)	0.10	0.01	0.53	-	5.04
RegNetX-400M	70.97	1.63 (1.63)	1.42 (2.65)	0.07	0.01	0.09	-	5.70
RegNetX-800M	74.04	1.12 (1.14)	0.97 (2.00)	0.19	0.02	0.24	-	4.38
RegNetX-1.6G	76.29	0.84 (0.85)	0.79 (1.88)	0.20	0.01	0.19	-	4.15
RegNetX-3.2G	77.89	0.61 (0.62)	0.53 (1.42)	0.20	0.00	0.24	-	3.70
EfficientNet-B0	76.83	0.75 (0.76)	1.70 (3.79)	0.15	0.03	0.19	-	4.39
EfficientNet-B1	78.13	0.57 (0.58)	1.18 (2.84)	0.26	0.01	0.39	-	3.26
EfficientNet-B2	79.97	0.57 (0.58)	1.13 (2.31)	0.05	0.04	0.41	-	3.10
EfficientNet-B3	82.03	0.71 (0.72)	0.99 (1.74)	0.16	0.05	0.38	-	2.65
EfficientNet-B4	83.43	0.29 (0.30)	0.45 (0.93)	0.17	0.02	0.26	-	2.32
ViT-Tiny	75.61	1.04 (1.04)	0.99 (1.79)	0.46	0.01	0.68	-	3.21
ViT-Small	81.58	0.57 (0.58)	0.37 (1.01)	0.80	-0.01	0.80	-	2.68
Vit-Base	84.63	0.61 (0.62)	0.43 (0.74)	0.93	-0.01	1.12	-	2.89
Swin-Tiny	81.32	0.18 (0.19)	0.42 (1.76)	1.21	0.00	0.76	-	4.93
Swin-Small	83.03	0.18 (0.18)	0.23 (1.33)	1.00	0.00	0.45	-	3.51
Swin-Base	83.54	0.11 (0.30)	0.21 (1.27)	0.97	-0.01	0.55	-	3.59

Table 2. On the contrary, the INT8 quantization increases its damage from 0.06% to 1.09%. This reveals two discoveries. First, different types of pre-processing noise can overlap with each other. Second, model inference noise might be magnified with other noises. We will provide more in-depth future studies. Interestingly, we show that the impact from SysNoise can be magnified especially in detection tasks whose model has ceil mode and upsample noise together. We deduce that this may be because they are both noises about the relative position and value of the model's feature map and the superposition of these two noises can cause effects beyond their own noise.

In Table 2, Table 3 and Table 4, we show how combined SysNoise affects different models on different model architecture. As shown in Table 2 and Table 3, adding all SysNoise to ResNet-50 together can damage 3.95% accuracy for classification and 10.67% mAP for detection, which equals degenerating a ResNet-50 lower than ResNet-34. Adding all SysNoise to EfficientNet-B4 makes it lower than the B3 variant. According to the original paper (Tan & Le, 2020), B4 consumes 2.3× more FLOPs than B3 and 1.6× higher parameters, yet only 1.4% accuracy improvement. However, SysNoise can easily make the architecture improvement useless, with up to 2.3% accuracy degradation to EfficientNet-B4.

4.3 Interpreting SysNoise

Does data augmentation improve model robustness against SysNoise? Studies have shown that data augmentation techniques can be used to improve model robustness against natural noises (Hendrycks et al., 2020; 2021b). We, therefore, employ these augmentation methods to see whether they could also improve the robustness against SysNoise. In particular, we train a ResNet-50 model using standard augmentation (He et al., 2015), APR-SP (Chen et al., 2021), Deepaug (Hendrycks et al., 2021a), AugMix (Hendrycks et al., 2020), and Deepaug combined with the other two methods (denoted "Deepaug+APR-SP" and

Table 3: **Measuring SysNoise on MS COCO detection.** We record mAP and the difference ΔmAP = $\text{mAP}_{\text{original}}$-$\text{mAP}_{\text{SysNoise}}$. We report both mean and max ΔmAP for decode and resize. *The lower* ΔmAP *the better*.

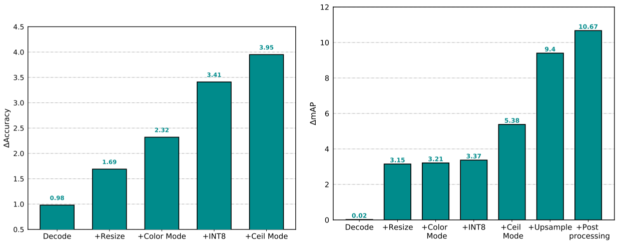
Method	**Architecture**	**Trained**	**Decode**	**Resize**	**Color Mode**	**Upsample**	**Precision INT8**	**Ceil Mode**	**Post-processing**	**Combined**
		mAP	ΔmAP	ΔmAP	ΔmAP	ΔmAP	ΔmAP	ΔmAP	ΔmAP	ΔmAP
Faster RCNN	ResNet-34	36.76	0.02 (0.04)	0.93 (2.63)	0.25	1.28	0.06	2.50	2.29	10.25
	ResNet-50	37.36	0.02 (0.01)	1.12 (3.15)	0.10	1.66	0.10	3.14	2.39	10.67
	MobileNetV2	30.32	0.01 (0.01)	0.38 (1.14)	0.24	0.96	0.07	-	2.23	3.45
RetinaNet	ResNet-34	35.71	0.01 (0.01)	0.77 (2.20)	0.29	0.35	0.10	2.72	3.44	8.21
	ResNet-50	36.59	0.01 (0.02)	0.99 (2.78)	0.36	0.69	0.03	3.12	3.00	8.93

Table 4: **Measuring SysNoise on CitySpace segmentation.** We record mIOU and the difference ΔmIOU = $\text{mIOU}_{\text{original}}$-$\text{mIOU}_{\text{SysNoise}}$. We report both mean and max ΔmIoU for decode and resize. *The lower* ΔmIOU *the better*.

Method	**Architecture**	**Trained**	**Decode**	**Resize**	**Color Mode**	**Upsample**	**Precision INT8**	**Ceil Mode**	**Combined**
		mIoU	ΔmIoU	ΔmIoU	ΔmIoU	ΔmIoU	ΔmIoU	ΔmIoU	ΔmIoU
DeepLabV3	ResNet-50	78.05	0.001 (0.001)	0.02 (0.04)	0.02	3.06	0.01	4.02	4.51
	ResNet-101	79.88	0.001 (0.001)	0.01 (0.02)	0.02	3.85	0.01	4.65	5.11
U-Net	-	61.98	0.003 (0.005)	0.04 (0.06)	0.04	2.74	0.02	-	2.85

Table 5: **Measuring SysNoise on Multiple NLP Datasets.** We record ACC on FP32 data precision and the difference ΔACC = $\text{ACC}_{\text{original}}$-$\text{ACC}_{\text{SysNoise}}$ on FP16 and INT8 data precision. *The lower* ΔACC *the better*.

Architecture	**PIQA**	**LAMBADA**	**HellaSwag**	**WINOGRANDE**
	FP32(ACC)/FP16(ΔACC)/INT8(ΔACC)			
OPT-125M	63.00/0.05/-0.06	37.90/0.04/0.37	29.20/0.01/0.15	50.28/0.00/-0.31
OPT-350M	64.36/-0.11/-0.33	45.16/0.00/-0.10	32.04/0.00/0.05	52.33/-0.08/0.24
OPT-1.3B	71.71/-0.05/0.16	58.06/0.07/0.19	41.45/-0.03/0.08	59.67/0.00/-0.24
OPT-2.7B	73.78/0.06/0.16	63.65/0.09/0.02	45.85/-0.01/0.01	61.01/0.00/0.00
OPT-6.7B	76.06/0.16/0.22	67.61/0.06/0.33	50.46/0.01/0.03	65.04/0.00/0.48
OPT-13B	75.90/0.11/0.06	68.72/0.07/0.29	52.44/-0.02/0.01	65.11/0.00/0.39
OPT-30B	77.69/0.11/0.16	71.47/0.02/0.00	54.29/-0.01/0.01	68.19/0.02/0.23



(a) ResNet-50 ImageNet (b) Faster-RCNN ResNet-50 COCO

Figure 3: **Illustration of the worst-case study by combining multiple SysNoise types step by step**.

"Deepaug+AugMix"). As shown in Fig. 4, we could reach several observations as follows (1) *there exist no single data augmentation methods that could universally achieve positive effects on all of the five different SysNoise types*; (2) specifically, data augmentations could improve model robustness against image decoder, the ceiling mode of the max-pooling layers (lower ΔACC). However, they fail to generalize for data precision and image resize (higher ΔACC). These indicate that SysNoise is highly diverse and inherently different from natural noises.

Does adversarial training improve model robustness against SysNoise? Besides natural noises, another axis to analyze SysNoise is to examine whether adversarially-robust models could be also effective against SysNoise. Here, we use adversarial training (the most effective method to defend adversarial noises) (Madry et al., 2018), and adversarially train ResNet-50 and RegNetX-3.2GF models with ℓ_∞-PGD attacks (Madry et al., 2018) using the standard setting (Tang et al., 2021; Croce et al., 2020). The results are summarized in Fig. 4, from which we can tell that *adversarial training has limited effect on improving model robustness against SysNoise* (significantly higher ΔACC on 80% SysNoise types). In some cases, like image decode and resize, adversarial training even significantly damages the model performance on SysNoise (significantly high ΔACC). Together with the data augmentation analysis, we show that SysNoise differs from both natural noises and adversarial noise, and the effectiveness of defenses that are designed for adversarial and natural noises is limited for SysNoise. We hope all these observations could inspire more in-depth future studies on building robust models against SysNoise.

Potential Methods to Improve Robustness against SysNoise. To solve SysNoise on the decoder and resize, a

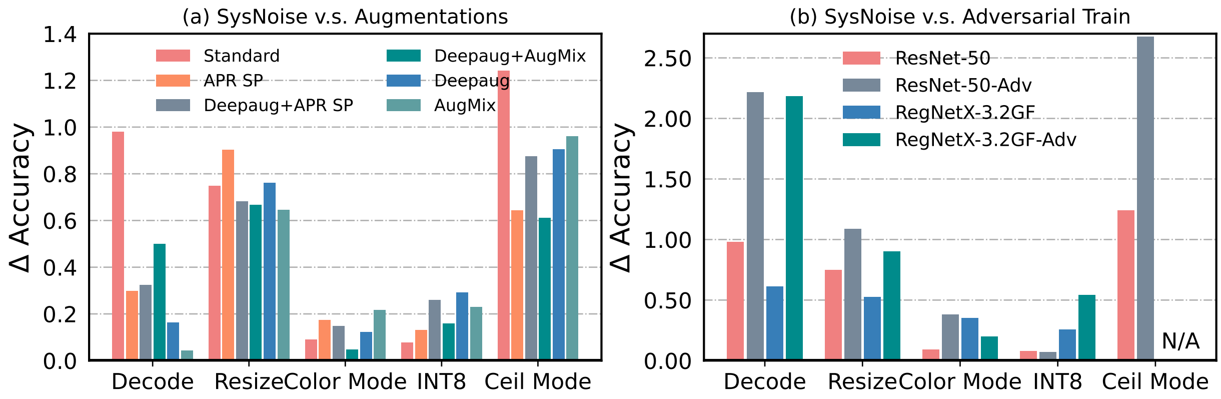


Figure 4: **Illustration of data augmentations and adversarial training for SysNoise on ImageNet**.

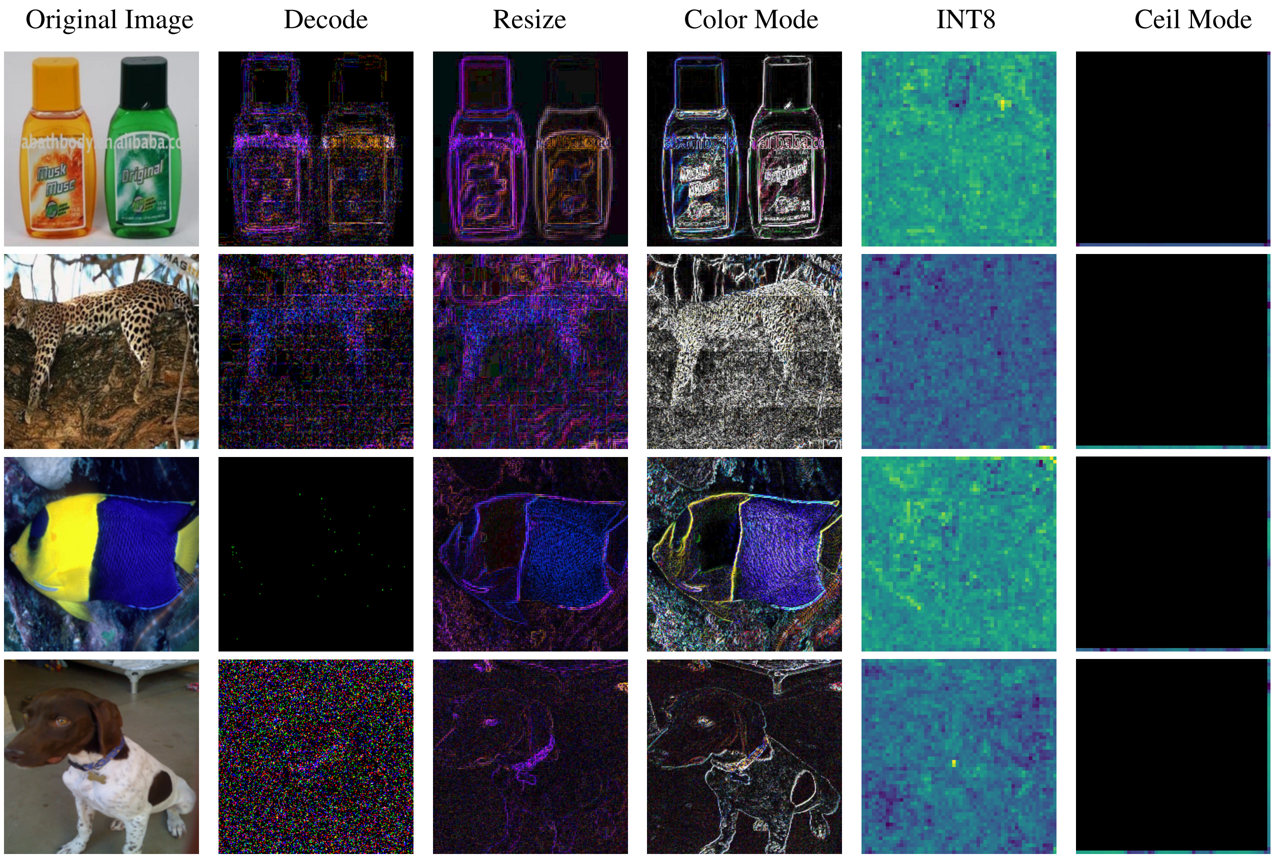


Figure 5: **Visualization of SysNoise. To make the noise more perceptible, we scale it to [0, 255]**.

Table 6: Mix training on resize method.

TrainTest	Pillow-bilinear	Pillow-nearest	Pillow-cubic	OpenCV-nearest	OpenCV-bilinear	OpenCV-cubic	Mean	Std.
Pillow-bilinear	76.572	72.168	76.512	72.090	75.346	74.072	74.460	2.02E+00
Pillow-nearest	74.872	75.988	75.548	75.970	76.002	76.056	75.739	4.63E-01
Pillow-cubic	76.312	72.828	76.596	72.876	75.810	74.666	74.848	1.68E+00
OpenCV-nearest	74.818	76.298	75.474	76.092	76.082	76.192	75.826	5.71E-01
OpenCV-bilinear	75.840	75.268	76.446	75.248	76.682	76.436	75.987	6.29E-01
OpenCV-cubic	76.194	72.812	76.510	72.940	75.736	74.818	74.835	1.62E+00
mix	76.154	75.876	76.344	75.786	76.444	76.330	**76.156**	**2.70E-01**

natural way is to make the model "see" all kinds of decoders and resize methods during the training process. Based on this principle, we introduce *mix training* method to enhance the model's robustness on system noise. The main process of mix training is to select the decoder or resize method randomly instead of just using one kind of method during the whole process of training. The pseudocode of our algorithm is shown in Algo. 1.

To test the effect of mix training, we set up the following experiment. We use ResNet50 as the base model of this experiment. To comprehensively demonstrate the training effect, we train single decoding and resize as well as our mix training models. We set the default decoder as Pillow and the default resize method as Pillow bilinear when conducting ablation studies on resizing method or decoder, respectively. Then we use top-1 accuracy as well as their mean and standard deviation as assessments. The results of this experiment are shown in Table 7 and Table 6. From these tables, we can conclude that: (1) The model has a better performance (usually the best) when we train and test using the same decoder and resize method. (2) Mix training can improve the robustness of a model on system noise greatly without hurting the clean accuracy. The $Std.$ using mix training drop from 0.36 to 0.0653 on decoder experiment, and drop from 0.463 to 0.270 on resize experiment. Meanwhile, it can maintain the model's accuracy at about 76%. In a contrast, the same ResNet50 model using $L_\infty - Robust$ adversarial training drops the $Std.$ from 1.07 to 0.420 by paying a 19.2% drop of clean accuracy.

Algorithm 1 Mixed training for improving robustness on systematic noise.

```
max width=0.48 *  Input: Resize set $\mathbb{RS}$; Decoder set $\mathbb{D}$;
        Model to train
Set Pillow-bilinear as default Resize;
 Set Pillow as default Decoder;
 for all $j = 1, 2, \dots, T$-iteration in training do
    if use mix-decoder strategy then
        Randomly sample a Decoder from $\mathbb{D}$;
    if use mix-resize strategy then
        Randomly sample a Resize function from $\mathbb{RS}$;
    Calling API to load the images from ImageNet-S according to the Decoder type and Resize type;
    Model Optimization.
return An optimized robust model for systematical noise.
```

Visualization. Here, we visualize the SysNoise by showing the *difference in pixels*. In specific, we calculate the differences between the clean image (or feature) and corrupted ones using SysNoise. As shown in Fig. 5, we can draw several interesting observations as follows. For the decode noise, it seems to be irregular (totally random or centered around the edge). As for resize and color mode noise, we observe that the differences often appear in edges or corners of an object (*i.e.*, shape). Specifically, Resize noise tends to mismatch in the red channel while color mode noise mismatches in all 3 channels. For Ceil Mode noise, it injects two bands of noises at the bottom right of the image. There is no obvious pattern for the INT8 noise.

Table 7: Mix training on the decoder.

TrainTest	Pillow	OpenCV	FFmpeg	Mean	Std.
Pillow	76.430	76.426	75.310	76.055	6.45E-01
OpenCV	76.510	76.510	75.368	76.126	6.56E-01
FFmpeg	75.730	75.664	76.318	75.904	3.60E-01
mix	**76.53**	**76.524**	**76.414**	**76.489**	**6.53E-02**

5 Conclusion

This paper introduces SysNoise, a harmful noise that frequently happens when the source training system switches to a disparate target system in deployments. We first identify and classify SysNoise based on the inference stage, and thereafter build a holistic benchmark and framework to quantitatively measure the impact of SysNoise on image classification, object detection, segmentation, and natural language processing tasks. Our large-scale experiments revealed that SysNoise is highly-influential and will cause model performance degeneration; additionally, common mitigations like data augmentation and adversarial training show limited effects on SysNoise.

In the future, we will evaluate SysNoise on the real-world systems, and will continuously develop the benchmark to include more tasks. Our findings open a new research topic and we hope it will raise research attention to the performance and robustness of deep learning deployment systems.

References

Bisk, Y., Zellers, R., Bras, R. L., Gao, J., and Choi, Y. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

Carlini, N. and Wagner, D. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pp. 39–57. IEEE, 2017.

Chen, G., Peng, P., Ma, L., Li, J., Du, L., and Tian, Y. Amplitude-phase recombination: Rethinking robustness of convolutional neural networks in frequency domain. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 458–467, 2021.

Chen, L.-C., Papandreou, G., Schroff, F., and Adam, H. Rethinking atrous convolution for semantic image segmentation. *arXiv:1706.05587*, 2017.

Chen, W.-H., Smith, C., and Fralick, S. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on communications*, 25(9):1004–1009, 1977.

Croce, F., Andriushchenko, M., Sehwag, V., Debenedetti, E., Flammarion, N., Chiang, M., Mittal, P., and Hein, M. Robustbench: a standardized adversarial robustness benchmark. *arXiv preprint arXiv:2010.09670*, 2020.

Cubuk, E., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. Autoaugment: Learning augmentation policies from data. In *IEEE CVPR*, 2018.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2021.

Gong, Z., Wang, W., and Ku, W.-S. Adversarial and clean data are not twins. *arXiv preprint arXiv:1704.04960*, 2017.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014a.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014b.

Grosse, K., Manoharan, P., Papernot, N., Backes, M., and McDaniel, P. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016a.

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

Hendrycks, D. and Dietterich, T. Benchmarking neural network robustness to common corruptions and perturbations. *Proceedings of the International Conference on Learning Representations*, 2019.

Hendrycks, D., Mu, N., Cubuk, E. D., Zoph, B., Gilmer, J., and Lakshminarayanan, B. Augmix: A simple data processing method to improve robustness and uncertainty, 2020.

Hendrycks, D., Basart, S., Mu, N., Kadavath, S., Wang, F., Dorundo, E., Desai, R., Zhu, T., Parajuli, S., Guo, M., et al. The many faces of robustness: A critical analysis of out-of-distribution generalization. In *ICCV*, 2021a.

Hendrycks, D., Zhao, K., Basart, S., Steinhardt, J., and Song, D. Natural adversarial examples. *CVPR*, 2021b.

HUAWEI. Huawei ascend cann. URL https://www.hiascend.com/en/software/cann.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.

Jia, K. and Rinard, M. Exploiting verified neural networks via floating point numerical error. In Drăgoi, C., Mukherjee, S., and Namjoshi, K. (eds.), *Static Analysis*. Springer International Publishing, 2021.

Jiang, W., He, Z., Zhan, J., and Pan, W. Attack-aware detection and defense to resist adversarial examples. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, Y., Shen, M., Ma, J., Ren, Y., Zhao, M., Zhang, Q., Gong, R., Yu, F., and Yan, J. MQBench: Towards reproducible and deployable model quantization benchmark. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=TUplOmF8DsM.

Liang, S., Wu, B., Fan, Y., Wei, X., and Cao, X. Parallel rectangle flip attack: A query-based black-box attack against object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7697–7707, 2021.

Lin, J., Chen, W.-M., Lin, Y., Gan, C., and Han, S. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33, 2020.

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., and Belongie, S. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2117–2125, 2017a.

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., and Belongie, S. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 936–944, 2017b. doi: 10.1109/CVPR.2017.106.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017c.

Liu, A., Liu, X., Fan, J., Ma, Y., Zhang, A., Xie, H., and Tao, D. Perceptual-sensitive gan for generating adversarial patches, 2019.

Liu, A., Liu, X., Yu, H., Zhang, C., Liu, Q., and Tao, D. Training robust deep neural networks via adversarial noise propagation. *IEEE Transactions on Image Processing*, 2021a.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., and Guo, B. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021b.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. Towards deep learning models resistant to adversarial attacks. *Proceedings of the International Conference on Learning Representations*, 2018.

NVIDIA. The nvidia data loading library. URL https://docs.nvidia.com/deeplearning/dali/user-guide/docs/.

Nvidia. Nvidia data loading library (dali). URL https://developer.nvidia.com/dali.

NVIDIA. Nvidia tensorrt. URL https://developer.nvidia.com/tensorrt.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N. Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. URL https://aclanthology.org/P16-1144.

Qualcomm. Qualcomm snapdragon neural processing engine. URL https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk.

Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., and Dollár, P. Designing network design spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10428–10436, 2020.

Rec, I. Bt. 601. encoding parameters of digital television for studios. *ITU, Geneva*, 1993.

Ren, S., He, K., Girshick, R., and Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. WINOGRANDE: an adversarial winograd schema challenge at scale. *CoRR*, abs/1907.10641, 2019. URL http://arxiv.org/abs/1907.10641.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Shafahi, A., Najibi, M., Ghiasi, A., Xu, Z., Dickerson, J., Studer, C., Davis, L. S., Taylor, G., and Goldstein, T. Adversarial training for free! *arXiv preprint arXiv:1904.12843*, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Su, D., Zhang, H., Chen, H., Yi, J., Chen, P., and Gao, Y. Is robustness the cost of accuracy? - A comprehensive study on the robustness of 18 deep image classification models. 2018.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

Tang, S., Gong, R., Wang, Y., Liu, A., Wang, J., Chen, X., Yu, F., Liu, X., Song, D., Yuille, A., Torr, P. H., and Tao, D. Robustart: Benchmarking robustness on architecture design and training techniques. *https://arxiv.org/pdf/2109.05211.pdf*, 2021.

Tomar, S. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006a.

Tomar, S. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006b.

Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I., Boneh, D., and McDaniel, P. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

Umesh, P. Image processing in python. *CSI Communications*, 23, 2012.

Wang, J., Liu, A., Yin, Z., Liu, S., Tang, S., and Liu, X. Dual attention suppression attack: Generate adversarial camouflage in physical world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8565–8574, June 2021.

Wikipedia. Converting yuv to rgb. URL https://en.wikipedia.org/wiki/YUV.

Wood, D. and Baron, S. Rec. 601—the origins of the 4: 2: 2 dtv standard. *EBU Technical Review and SMPTE Journal*, 2005.

Xiang, L., Shouling, J., Jiaxu, Z., Jiannan, W., Chunming, W., Bo, L., and Ting, W. Deepsec: A uniform platform for security analysis of deep learning model. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

Yinpeng, D., Qi-An, F., Xiao, Y., Tianyu, P., Hang, S., Zihao, X., and Jun, Z. Benchmarking adversarial robustness on image classification. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *CoRR*, abs/1905.07830, 2019. URL http://arxiv.org/abs/1905.07830.

Zhang, H., Cissé, M., Dauphin, Y. N., and Lopez-Paz, D. mixup: Beyond empirical risk minimization. *CoRR*, abs/1710.09412, 2017. URL http://arxiv.org/abs/1710.09412.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022. URL https://arxiv.org/abs/2205.01068.

A Mathematical Difference of SysNoise

In this section, we try to use equations to describe how different processing, operations are formulated. Note that our explanation might not be exactly the same with third-party implementations, as there are always some hyper-parameters to determine. Our goal is to provide an intuition rather than a strict comparison.

Image Decode. In the decoding process, the inverse discrete cosine transform (iDCT) occupies the majority of the computation. Given a transformed matrix $\hat{\mathbf{X}}$ with shape $N \times N$ (excluding channels), the original image $\mathbf{X}$ at coordinates (m, n) can be given by

$$f[m,n] = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1}\alpha(k)\alpha(l)F(k,l) \\ cos[\frac{(2m+1)\pi k}{2N}]cos[\frac{(2n+1)\pi l}{2N}] \tag{1}$$

where,

$$\alpha(k) = \begin{cases} \sqrt{\frac{1}{N}} \text{ if } k = 0 \\ \sqrt{\frac{2}{N}} \text{ if } k \neq 0 \end{cases}. \tag{2}$$

The iDCT costs a lot of operations and some implementations choose to utilize Fast DCT and Fast iDCT (Chen et al., 1977) where the computation is sped up by matrix decomposition. Due to its complexity, we do not display the equations here. Note that the de-quantization in decode will also bring different values, which will be introduced in the data precision section.

Resize Interpolation. Formally, considered an image $\mathbf{X}$ to be resized where a pixel in some position needs to be predicted and yet its neighbors are already known or predicted. Different interpolation algorithms rely on different functions to determine the unknown pixel. (1) Nearest interpolation, this method simply copy the nearest neighbor's pixel value, *i.e.*, the neighbor with the lowest Euclidean distance, given by $\mathbf{X}[\arg\min_{x,y}((x-x')^2+(y-y')^2)]$. Here, the x, y is the coordinates of the known neighbor and x', y' is the coordinates of the pixel that needs to be determined. (2) Bilinear interpolation, determines the pixel by linearly calculating the ratio of distance. Assume we have four spatially-close coordinates: $Q_{11} = (x1, y1)$, $Q_{12} = (x1, y2)$, $Q_{21} = (x2, y1)$, and $Q_{22} = (x2, y2)$. Their values are already know, for example $f(Q_{11})$. The formulation of bilinear interpolation is given by:

$$f(x,y) = \frac{y_2 - y}{y_2 - y_1}f(x,y_1) + \frac{y - y_1}{y_2 - y_1}f(x,y_2), \tag{3}$$

where,

$$f(x,y_1) = \frac{x_2 - x}{x_2 - x_1}f(Q_{11}) + \frac{x - x_1}{x_2 - x_1}f(Q_{21}),$$
$$f(x,y_2) = \frac{x_2 - x}{x_2 - x_1}f(Q_{12}) + \frac{x - x_1}{x_2 - x_1}f(Q_{22}).$$

(3) Bicubic interpolation, in contrast to the bilinear interpolation which only takes 4 pixels (2×2), the bicubic interpolation takes 16 pixels (4×4). The algorithm tries to use existing known pixel values to fit a binary cubic function

$$f(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j \tag{4}$$

To find the total 16 coefficients $a_{ij}, ij \in \{0,1,2,3\}$, we need to solve a system of linear equations $A\alpha = x$. Due to the complexity of this algorithm, we refer the readers to this link[1] for more details. Bicubic interpolation yields better performance than the previous two algorithms, however, it also needs huge time to solve the linear equations to find optimal interpolated values. We omit other interpolations methods as they are more complex that these three methods.

YUV color mode. As a matter of fact, there are tons of encoding standards for YUV color space. The formats described here all use 8 bits per pixel location to encode the Y channel (also called the luma channel), and use 8 bits per sample to encode each U or V chroma sample. However, most YUV formats use fewer than 24 bits per pixel on average, because they contain fewer samples of U and V than of Y. The full-size YUV (32 bits per pixel) is represented as 4:4:4, which means no downsampling of chroma channels. Following BT.601 (Rec, 1993), converting RGB to YUV 4:4:4 can be formulated by

$$\begin{cases} Y = \text{round}(0.256788 \times R + 0.504129 \times G + 0.097906 \times B) + 16 \\ U = \text{round}(-0.148223 \times R - 0.290993 \times G + 0.439216 \times B) + 128 \\ V = \text{round}(0.439216 \times R - 0.367788 \times G - 0.071427 \times B) + 128 \end{cases}. \tag{5}$$

Here, we can derive an inverse transform from YUV to RGB,

$$\begin{cases} R = \text{clip}(\text{round}(1.164383 * C + 1.596027 * E)) \\ G = \text{clip}(\text{round}(1.164383 * C - (0.391762 * D) - (0.812968 * E))) \\ B = \text{clip}(\text{round}(1.164383 * C + 2.017232 * D)) \end{cases}, \text{where} \begin{cases} C = Y - 16 \\ D = U - 128 \\ E = V - 128 \end{cases}. \tag{6}$$

Here, $\text{clip}(\cdot)$ denotes clipping to a range of $[0, 255]$. In some implementation (Wikipedia), Eq. (6) can be approximated by:

$$\begin{cases} R = \text{clip}((298 * C + 409 * E + 128) >> 8) \\ G = \text{clip}((298 * C - 100 * D - 208 * E + 128) >> 8) \\ B = \text{clip}((298 * C + 516 * D + 128) >> 8) \end{cases}. \tag{7}$$

[1] https://www.ece.mcmaster.ca/~xwu/interp_1.pdf

As we could see, the conversion cannot be lossless with the existence of rounding and clipping operations, which could be generally summarized to *quantization-dequantization* conversion. In addition, usually, the hardware supports YUV 4:2:0 rather than 4:4:4, making the conversion to RGB more unstable cause YUV 4:2:0 should be transformed to YUV 4:4:4 and then transformed to RGB format (Wood & Baron, 2005).

Ceiling mode. For pooling layers, the output shape of the feature map is calculated by

$$O = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1, \tag{8}$$

where W is the width (we assume the feature map is square), K is the kernel size, P is the padding size, and S is the stride of pooling layers. The above equation uses floor operation $\lfloor \cdot \rfloor$ to compute the size of the output feature while we can use ceiling operation $\lceil \cdot \rceil$ operation in ceiling mode. Therefore, the border of the output feature is dependent on the ceiling mode.

Data Precision. We here discuss two types of precision: FP16 and INT8. The FP16 still uses floating-point numbers with less bitwidth. According to IEEE 754, the FP32 format uses 1 bit for sign, 8 bits for the exponent, and the rest 23 bits for fraction, while the FP16 uses 1 bit for sign, 5 bits for the exponent, and 10 bits for fraction. Normally, converting FP32 to FP16 only causes a negligible error, as shown in our experiments. For INT8, this is usually done by quantization and de-quantization functions:

$$\bar{\mathbf{X}} = \text{clip}\left(\lfloor \frac{\mathbf{X}}{s} \rceil + z, N_{min}, N_{max} \right) \tag{9}$$

$$\hat{\mathbf{X}} = s * (\bar{\mathbf{X}} - z), \tag{10}$$

where $\lfloor \cdot \rceil$ is the rounding-to-nearest function. N_{min}, N_{max} are the range of integers that can be represented. For INT8, $N_{min} = -128$ and $N_{max} = 127$. $s \in \mathcal{R}$ and $z \in \mathcal{Z}$ are the scale and zero point parameters to fit the original FP32 tensor's range. For more details of quantization, readers are recommended to (Li et al., 2021).

Post-processing. For object detection, the post-processing involves multiple operations: 1. calculate the anchors, 2. get the offsets for anchors from the predicted outputs, 3. calculate the final bounding box. Some details of these operations are easy to bring the noise. Some details of these operations are easy to cause noise. The following code shows an example procedure for post-processing. For different hardware implementations, the **ALIGNED_FLAG.offset** in the code often has different values of 0 or 1. This minor difference will bring a perturbation to the final accuracy performance. Besides, other operations like the rounding from float-point output to integer coordinate or the precision of exponential also need to be treated carefully.

```
# anchors from xyxy format to xywh
   format
ctr_x, ctr_y, widths, heights =
   xyxy2xywh(boxes)

# normalize the offsets predicted from
   the neural network
means = offset.new_tensor(means).view
   (1, -1).repeat(1, offset.size(-1)
   // 4)
stds = offset.new_tensor(stds).view(1,
   -1).repeat(1, offset.size(-1) // 4)
offset = offset * stds + means

# calculate the delta of x, y, w and h
wx, wy, ww, wh = weights
dx = offset[:, 0::4] / wx
dy = offset[:, 1::4] / wy
dw = offset[:, 2::4] / ww
dh = offset[:, 3::4] / wh

dw = torch.clamp(dw, max=np.log(1000. /
    16.))
dh = torch.clamp(dh, max=np.log(1000. /
    16.))

# calculate the predicted coordinate of
    center point,
# and the height & weight of bbox
pred_ctr_x = dx * widths[:, None] +
   ctr_x[:, None]
pred_ctr_y = dy * heights[:, None] +
   ctr_y[:, None]
pred_w = torch.exp(dw) * widths[:, None
   ]
pred_h = torch.exp(dh) * heights[:,
   None]

# calculate the final bbox
pred_boxes = offset.new_zeros(offset.
   shape)
# x1
pred_boxes[:, 0::4] = pred_ctr_x - 0.5
   * pred_w
# y1
pred_boxes[:, 1::4] = pred_ctr_y - 0.5
   * pred_h
# x2
pred_boxes[:, 2::4] = pred_ctr_x + 0.5
   * pred_w - ALIGNED_FLAG.offset
# y1
pred_boxes[:, 3::4] = pred_ctr_y + 0.5
   * pred_h - ALIGNED_FLAG.offset
```

B Broader Impacts and limitations

Together with existing benchmarks on adversarial and natural noises, we could build a more comprehensive and general understanding and ecosystems for robustness benchmarking involving more perspectives. We hope this benchmark could draw the attention of both algorithm researchers and hardware vendors to this inevitable and urgent-to-solve problem, and open a new research direction for building robust deep learning deployment systems.

Though having investigated several types of SysNoise in this paper, there may still exist other noises that would cause model performance degeneration during deployment. In the future, we will keep the benchmark growing.

C Consistency of Results

To maintain consistency of results, we use following method. (1) Fix in the requirements torch==1.8.1, opencv==4.1.1.26 and Pillow==6.2.1 in our framework. (2) Set torch.backends.cudnn.benchmark=True in the code. We test the ResNet-18 Model on all kinds of noise multiple times in our framework and observe little different result ($< 0.0001\%$) on accuracy. This result also holds for object detection and instance segmentation task. So other factors are less likely to affect the results of the model inference process.

D Reproducibility and Run Time

We provide the code to run this benchmark on GitHub where everyone can download from freely. As for the setup steps and instructions about our code, we provided them in the README file. The installation instructions are also provided in the README file, users can easily install the required run time environment of this codebase. For some noises that need to be generated on specific hardware and are not easy to reproduce, we provide our own resulting datasets generated on specific hardware, which involve ImageNet validation set and COCO validation set. All these datasets can be freely downloaded on our website.

Since our benchmark experiments need us to train multiple models and evaluate them on different kinds of noises, it needs a large amount of GPU resources. The total cost of our GPU resources to build this benchmark is about 5 GPU years. Most of our experiments are run on Nvidia Tesla V100 GPU. For one training experiment, we run it on 16 GPUs parallel. For inference experiments, we run it on 4 GPUs parallel.

For other users who just want to test their trained model with our framework, the GPU time they require will be greatly reduced. In most cases it only takes 10 to 40 minutes of GPU time to test the effect of one noise on one model, depending on the GPU type they are using.

E License

Our code is released under Apache License 2.0. Most model architectures are added to the code with the license chosen by the original author. The ImageNet-1K, COCO, and CitySpace datasets we use are downloaded from the official release. Some system noise datasets we generated from the original dataset follow the license of its original dataset.