





Identifying, Evaluating, and Addressing Nondeterminism in Mask R-CNNs

Stephen Price^(✉)  and Rodica Neamtu 

Computer Science Department, Worcester Polytechnic Institute, Worcester, MA, USA
{sprice,rneamtu}@wpi.edu

Abstract. Convolutional Neural Networks, and many other machine learning algorithms, use Graphical Processing Units (GPUs) instead of Central Processing Units (CPUs) to improve the training time of very large modeling computations. This work evaluates the impact of the model structure and GPU on nondeterminism and identifies its exact causes. The ability to replicate results is quintessential to research, thus nondeterminism must be either removed or significantly reduced. Simple methods are provided so that researchers can: (1) measure the impact of nondeterminism, (2) achieve determinable results by eliminating randomness embedded in the model structure and performing computations on a CPU, or (3) reduce the amount of variation between model performances while training on a GPU.

Keywords: Nondeterminism · Mask R-CNN · Determinable · GPU · NVIDIA · Computer vision · Embedded randomness · Replicability

1 Introduction

Recent advancements in computer vision using Convolutional Neural Networks (CNNs) have emphasized their ability to classify images and detect objects within images [16]. However, these tasks require significant computational resources and time to complete [25]. Fortunately, Graphics Processing Units (GPUs) are more capable of handling these advanced computations than Central Processing Units (CPUs) [9, 27]. For example, models trained in this work require 20–30 min to complete on a GPU but take 10–14 h on a CPU. GPUs were originally configured to work with graphics. However, as researchers showed more interest in using GPUs for computations associated with machine learning, NVIDIA (the most commonly used GPU manufacturer for machine learning [20]) created the CUDA Library [12], thus enabling the use of GPUs for diverse machine learning tasks. Unfortunately, NVIDIA’s GPU and CUDA library introduce nondeterminism reflected in two or more identically trained models sometimes producing different results. This GPU-related nondeterminism is distinct from the nondeterminism due to randomness embedded in the model structure by features such as Stochastic Data Augmentation and Stochastic Weight Initialization. The existence and impact of nondeterminism related to both the randomness embedded in the model structure and the GPU have gained increasing attention [20], and deserve continuous assessment and research to reduce it.

1.1 Motivation

This work is inspired and motivated by previous research [22] using a Mask R-CNN [28] to analyze metallic powder particles and detect deformations on the surface. Since these deformations known as satellites, impact the usability of metallic powders, accurate detection is very important. In the previous work, using a Mask R-CNN led to accurate detection, even on a diverse dataset composed of multiple powder types taken at varying magnification settings. However, upon deep analysis of results looking for determinism, it was discovered that, in some cases, two or more identically trained models could produce significantly different results. Figure 1, depicting two identically trained models, labeled Model A and B, highlights these differing results due to nondeterminism. These models were specifically selected to illustrate potential variation in outputs. In Fig. 1, the outlined green section highlights a small satellite detected in Fig. 1a more than tripling in size and losing its discernible shape in Fig. 1b. Similarly, in the section outlined red, two particles correctly identified to have no satellites in Fig. 1a are misidentified as satellites in Fig. 1b. Inspired by the nondeterminism causing these variations, this manuscript aims to quantify its impact and provide viable options to reduce or remove it to ensure replicability of experimental results.

1.2 Terminology

To avoid confusion due to various existing definitions, for the purpose of this work, determinism and nondeterminism are defined as follows: (1) An algorithm is said to be “determinable” if its current state uniquely determines its next state. Simply put, an algorithm at any state should produce exactly one output. (2) An algorithm is said to be “nondeterminable” if, at a given state, multiple potential outputs are possible given the same input [14]. In the context of this paper, *if all models trained within a given environment are identical, that training environment is determinable.*

1.3 Contributions

The ability to replicate results is quintessential to research [10]. Thus, being able to eliminate, or at least reduce nondeterminism in a Mask R-CNN is imperative. The contributions of this work are multi-fold:

1. Identifying and evaluating the causes and extent of nondeterminism in Mask R-CNN models with embedded randomness trained on an NVIDIA GPU.
2. Evaluating the extent of nondeterminism in Mask R-CNN models with no embedded randomness trained on an NVIDIA GPU.
3. Offering a simple method, requiring only eight additional lines of code, to achieve pure deterministic results through a combination of using a CPU and specific training configurations.

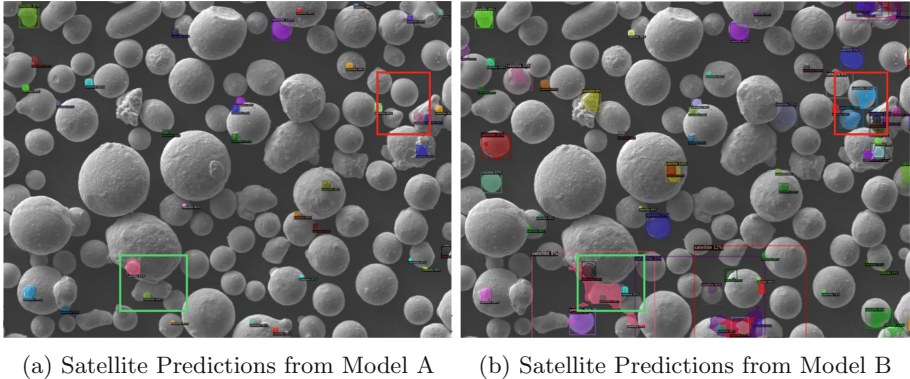


Fig. 1. Example of variation of performance in identically trained Mask R-CNN models as a result of nondeterminism (Color figure online)

2 Nondeterminism Introduced in Training

To measure nondeterminism in model training caused by GPUs, all other sources of nondeterminism must first be eliminated. Through rigorous examination of literature, documentation, and user manuals of the varying tools and packages [12, 21, 28], the following have been identified as potential sources of nondeterminism embedded in the model: *Random Number Generators* (used by Python Random Library, PyTorch, NumPy, and Detectron2), *Detectron2 Augmentation Settings*, and the *PyTorch implementation of CUDA Algorithms*. Figure 2 illustrates the general sources of nondeterminism that may be present in a Mask R-CNN, as well as the components of each source. The following subsections give some background information on each of these sources.

2.1 Random Number Generators

The training of a CNN employs randomness for large-scale computations to reduce training time and prevent bottlenecks [5, 29]. Each instance of embedded randomness is enabled by a Pseudo-Random Number Generator (PRNG) that generates sequences of numbers designed to mimic randomness. Mersenne Twister (MT) [19] is one of the most frequently used PRNG algorithms by tools such as the Python Random Library [26]. MT simulates randomness by using the system time to select the starting index or seed in the sequence of numbers when a PRNG is created [26]. Without a set seed, each PRNGs starts at a unique index, leading to different outputs and introducing nondeterminism in training.

2.2 Model Structure

This model structure is configured by Detectron2 [28], which uses a fairly common training technique called Stochastic Data Augmentation to randomly mirror

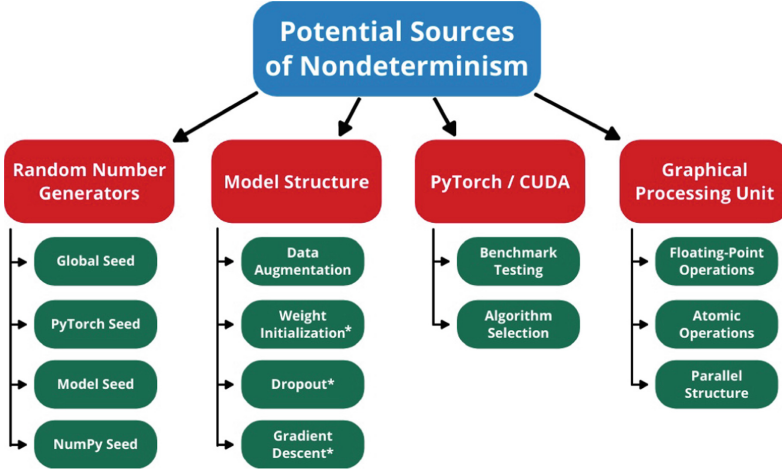


Fig. 2. General sources of nondeterminism that may be found in Mask R-CNNs (*not present in this work, but may be present in other implementations)

images prior to training [23]. The stochastic process of selection increases the nondeterminism in training. Augmentation is the only source of nondeterminism caused by the model structure here. However, this may not always be the case.

2.3 CUDA Algorithms and PyTorch

The PyTorch implementation of the CUDA Library [21], by default, contains two settings that increase nondeterminism in training. First, CUDA uses Benchmark Testing to select optimal algorithms for the given environment. However, as indicated in the documentation [21], this testing is “susceptible to noise and has the potential to select different algorithms, even on the same hardware.” Second, by default, the library chooses nondeterminable algorithms for computing convolutions instead of their determinable counterparts. These nondeterminable algorithms are selected because they simplify computations by estimating randomly selected values instead of computing exact values for each layer [6]. Both configurations increase the nondeterminism present in model training.

3 Nondeterminism Introduced by Hardware

3.1 Floating-Point Operations

Many computer systems use floating-point numbers for arithmetic operations; however, these operations have a finite precision that cannot be maintained with exceptionally large or small numbers. In this work, values were stored using the IEEE 754 Single Precision standard [18]. Unfortunately, due to the finite

precision of floating-point numbers, some calculations are approximated, causing a rounding error and rendering the associative property not applicable in floating-point operations [15]. Equations 1 and 2 provide an example in which the non-associativity of floating-points impacts the final result. In the intermediate sum in Equation 1, the 1 is rounded off when summed with 10^{100} , causing it to be lost in approximation. When computing the difference after rounding, $10^{100} - 10^{100}$ returns **0**. By contrast, if $10^{100} - 10^{100}$ is performed first, the result of computations will be 0 at the intermediate step, and when summed with 1, will return the correct value of **1**. In summary, due to the non-associativity of floating-points, the order in which operations are executed impacts the outputs. This becomes increasingly relevant when parallel computing is implemented, as further elaborated in Sects. 3.2 and 3.3.

$$(1 + 10^{100}) - 10^{100} = \mathbf{0} \quad (1) \qquad 1 + (10^{100} - 10^{100}) = \mathbf{1} \quad (2)$$

3.2 Atomic Operations

Shared memory is commonly implemented in parallel computing [3]. However, when multiple operations access the same location in memory at similar times, depending on when read and write methods are called, data can be “lost” due to overlapping operations [11]. Atomic Operations resolve this by performing a read and write call as an atomic action and preventing other operations from accessing or editing that location in memory until completed. Atomic operations are designed to ensure memory consistency but are unconcerned with the completion order consistency [11]. Effects of this are noted in the CUDA Library [12], stating “the following routines do not guarantee reproducibility across runs, even on the same architecture, because they use atomic operations” in reference to a list of algorithms used in convolutions.

3.3 Parallel Structure

With the introduction of the CUDA library, taking advantage of the benefits of parallel computing with GPUs became easier [8] and more frequently used [13]. Despite these benefits, there are inherent drawbacks to most multi-core or multi-threaded approaches. In parallel computing, large computations are broken into smaller ones and delegated to parallel cores/threads. Each sub-task has a variable completion time, which is amplified by the use of atomic operations. When considering the variable completion time of various tasks and the non-associativity of floating-point operations, it is not surprising that GPUs introduce nondeterminism. Figure 3 illustrates how a slight variation in the completion order of sub-tasks can lead to nondeterminable results due to floating-point operations. Figures 3a and 3b depict the process `sum()` adding sub-functions (labeled F1 to F5) together, in which each sub-function is dispatched to its own core/thread to be individually computed. The output (labeled O1 to O5) is collected in

order of completion and summed. However, since completion order is not guaranteed, these outputs can be collected in different orders, resulting in differing output despite having identical inputs and hardware because of floating points non-associativity. [15].

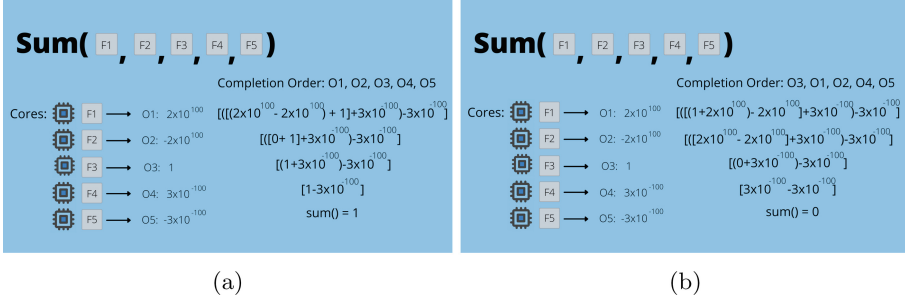


Fig. 3. Tracing the impact of variable completion times in parallel structures using floating-point operations

4 Experimental Setup

This work used the Detectron2 implementation of the Mask R-CNN with PyTorch (v1.8.1) [21] and CUDA (v10.2) [12]. Initial weights were pulled from the Detectron2 Model Zoo Library to remove any variation in weight initialization. The dataset used here is the same dataset used in [22], consisting of images of metallic powder particles collected from a Scanning Electron Microscope. It contains 1,384 satellite annotations across six powders and five magnifications and was separated using an 80:20 ratio between training and validation datasets.

4.1 System Architecture

To get a benchmark for variation in performance caused by embedded randomness in the model structure and the GPU, 120 models were trained using an NVIDIA V100 GPU. Of these models, 60 were left non-configured and 60 were configured, as shown in the source code [2], such that all embedded randomness within the model structure was disabled. This ensured that any nondeterminism present after configuring models was induced solely by the GPU. This experiment was then replicated using CPUs. However, due to the high time difference in training between GPUs and CPUs, a 5-Fold Cross Validation [4] was used instead of training 60 models. For example, based on results from the ten models trained on the CPU, training 120 models on a CPU would have taken between 50 and 70 days of computational time, instead of 48 h on a GPU. The 5-Fold Cross Validation was used only to evaluate if results were determinable over multiple iterations; due to the small number of data points it was not used to evaluate the extent of nondeterminism.

4.2 Measuring Performance

Identical to previous work [22], performance was measured by computing precision and recall, as defined in Equations 3 and 4. For every image in the validation set, each pixel was classified as True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) depending on its true value and the predicted output. Once these scores were computed for each image, they were averaged across all images in the validation set to get a final score for that model to be compared. Nondeterminism was evaluated by analyzing the average, standard deviation, and spread of each performance metric collected by models trained with identical configuration settings. If the training configurations and hardware are determinable, precision and recall will be identical for all models trained.

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

4.3 Model Training Process

Previous work [22] discovered that in most cases, training beyond 10,000 iterations had little impact on performance. As a result, in an effort to prevent underfitting or overfitting, all models were trained to 10,000 iterations. Additionally, to prevent introducing any bias, hyperparameters were left at their default values. All calculations were completed in batch jobs dispatched to private nodes on Bridges2 [7], a High-Performance Computer (HPC) operated by Pittsburgh Supercomputing Center and funded by the National Science Foundation. Each node contained two NVIDIA Tesla V100 GPUs and two Intel Xeon Platinum 8168 CPUs.

4.4 Configuring Settings

To compare models with embedded randomness enabled and respectively disabled, specific configurations had to be set. Table 1, depicting configuration settings, shows the value of each configuration for models with and without embedded randomness. Configuring an RNG’s seed only changes the starting index and has no further impact on the randomness [19]. As a result, so long as the seed remains constant, its specific value is arbitrary. Evidence for this is found in the Detectron2 Source Code stating the seed needs to be “any positive integer” and in the NVIDIA Determinism Repo stating “123, or whatever you choose” [1, 28]. In light of this, all seeds were arbitrarily set to “42.” After reviewing the CUDA Toolkit Documentation [12], the PyTorch Documentation [21], the Detectron2 Source Code [28], and NVIDIA Determinism Repository [1], the only possibility for achieving reproducible results that was not implemented was the PyTorch DataLoader. This was not configured because Detectron2 implements its own custom DataLoader class and the PyTorch version was not used.

Table 1. Configuration values for non-configured and fully configured models

Configuration name	Non-configured value	Fully configured value
Data augmentation	Horizontal	None
Benchmark testing	True	False
Determinable algorithms	False	True
RNG seeds	NA	42

5 Experimental Results

5.1 Data Collected from Models Trained on GPU

After training 120 models on a GPU (60 non-configured and 60 fully configured), regardless of configuration settings, there was clear evidence of nondeterminism. Table 2 shows all performance metrics gathered from models trained on a GPU for comparison, but attention will be drawn specifically to the standard deviation of precision and recall values (bolded and marked * and ** respectively). As can be seen in Table 2, configuring the embedded randomness in the model decreased the standard deviation of precision values by 1% (marked *) and recall by 0.1% (marked **). Despite the 1% reduction in variation of precision, only 25% of the nondeterminism is eliminated, leaving a remaining 3.1% standard deviation caused by the GPU. Figure 4 shows the distributions of precision values for non-configured and fully configured models. As can be seen, the distribution of Fig. 4a, corresponding to non-configured models, has a larger spread of data points than in Fig. 4b, corresponding to fully configured models.

Table 2. Performance metrics for non-configured and fully configured models

Model performance metrics on GPU	Non-configured	Fully configured
Average precision (%)	72.908%	70.750%
Precision std deviation (%)	4.151%*	3.135%*
Precision range (min:max)	63.5% : 82.7%	64.4% : 78.1%
Average recall (%)	61.581%	60.914%
Recall std deviation (%)	2.733%**	2.603%**
Recall range (min:max)	54.4% : 67.6%	55.8% : 66.5%
Average training time (min)	19.497	28.757
Training time std deviation (min)	0.359	0.304
Training time range (min:max)	19.0 : 21.32	28.25 : 29.5

5.2 Data Collected from Models Trained on CPU

Since only five models were trained on a CPU instead of 60 due to the expected very large training time as previously discussed, the presence or absence of nondeterminism can be observed but not quantified. Models trained on a CPU with all embedded randomness disabled produced perfectly determinable results. These results were identical up to the 16th decimal place (only measured to 16 decimal places) with a precision score and recall of approximately 76.2% and 56.2%, respectively. Among these models, there was a minimum training time of 606.15 min, a maximum of 842.05 min, and an average of 712.93 min. The variety in training times had no impact on the accuracy of the model. In contrast to every trained model with no embedded randomness on the CPU having identical precision and recall score, when embedded randomness was enabled, there wasn't a single duplicated value. As shown in Table 3 depicting a comparison of precision and recall scores of non-configured models trained on a CPU, each model produced quite different results, showing that nondeterminism is present. As a result of nondeterminism being present in CPU trainings when embedded randomness is enabled, nondeterminism can, in part, be attributed to embedded randomness in the model.

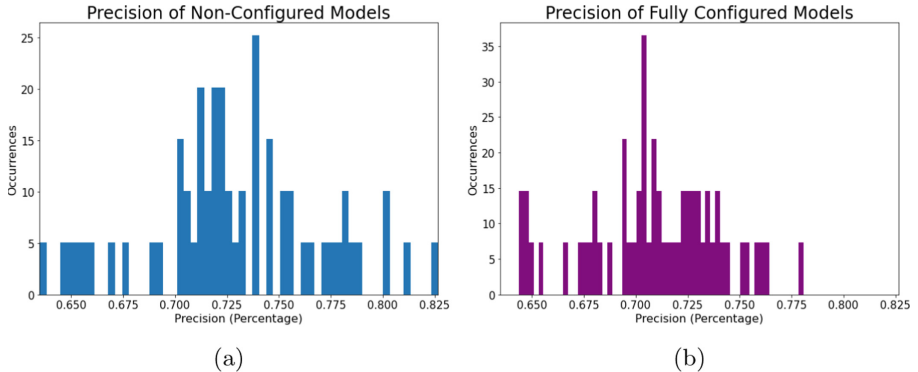


Fig. 4. Comparative results of the distribution of precision values collected from non-configured models (a) and fully configured models (b) trained on a GPU

Table 3. Performance metrics for non-configured models trained on CPUs

Training num	Precision (%)	Recall (%)	Time (min)
Model 1	0.7992862217337212	0.6222949769133339	646.07
Model 2	0.7321758705343052	0.6506021842415332	720.83
Model 3	0.7019480667622756	0.6118534248851561	606.23
Model 4	0.7585633387152428	0.6253458176827655	612.05
Model 5	0.7534353070389288	0.6330197085802262	616.65
Average	0.7490817609568948	0.628623222460603	640.37
Std deviation	0.0320522567741504	0.012918343823211	42.51

6 Discussion of Results

6.1 Impact of Embedded Randomness on Model Precision

As previously shown, randomness is deliberately embedded in machine learning models to improve their generalizability and robustness [17]. By eliminating the embedded randomness within the model, there is an associated reduction in the ability of the model to generalize for samples of data with more variation than those within the training set. In context, by decreasing the randomness embedded in the model structure during training, the model’s ability to handle formations of satellites not included in the training set may decrease. This could explain why the average precision and recall values were lower in the fully configured model, and why there was a reduced number of models with precision scores above 75% out of the set of fully configured models (4 models) compared to the non-configured models (17 models). In summary, *by disabling embedded randomness, the model may be less capable of handling new data, and as a result, it may be less generalizable.*

6.2 Increase in Training Time After Configuring Randomness

Even though the reduction in performance variation was about 25% after disabling the randomness embedded in the model structure, the training time increased by nearly 50%. Non-Configured models took on average 19.5 min to train on a GPU, which rose to 28.8 min after configuring the embedded randomness. This increase was theorized to be the result of forcing CUDA algorithms to be determinable instead of their nondeterminable counterparts. To test this, 40 models were trained with all embedded randomness disabled except Determinable Algorithms. With these parameters, the models had nearly identical precision and recall scores to a fully configured model with an average score of 71.966% and 60.207% respectively, and standard deviations of 3.455% and 2.354%. However, the average training time decreased to 19.1 min with a standard deviation of 0.130 min, much closer to that of the non-configured models. As a result, *since forcing determinable algorithms has a minimal impact on the variation but increases the training time by approximately 50%, it is suggested to allow nondeterminable algorithms when response time is a priority.*

6.3 Impact of Seed Sensitivity

By disabling embedded randomness within the model structure, there was little adverse impact on performance. Between non-configured models and fully configured models on the GPU, precision and recall were reduced on average by 2% and 0.6%, respectively. Since each seed outputs different values than another seed and slightly impact performance, the model is seed-sensitive [24]. In this case, the seed was arbitrarily set to “42.” However, other seed values may produce different results. Thus, *if hyperparameter tuning is being performed with a configured seed, users may consider testing multiple seed values to identify which works best for the given dataset and parameters.*

6.4 Conclusion

The methods and procedures highlighted in this manuscript aim to inform the selection process of parameters and hardware for training a Mask R-CNN model with respect to nondeterminism and training time. In cases where determinable results are of a priority, model training can be performed on a CPU with the embedded randomness in the model structure configured. This will guarantee fully determinable results and only requires an additional eight lines of code. These configurations can be found in the training files for the repository associated with this manuscript [2]. Unfortunately, by running computations on a CPU instead of a GPU, the training time increases from 20–30 min to 10–14 h. As a result, a CPU should only be used in cases where computational resources are not a concern and replicability is more important than speed and efficiency. If determinable results are not the first priority, in most cases performing training on a GPU is a better choice. However, in addition to the reductions in training time accomplished by using a GPU (at least 20 times faster), the nondeterminism present during model training will increase. Here, the standard deviation of this variation in non-configured models was approximately 4.2% and 2.7% for precision and recall respectively. Using the methods established above, this variation can be reduced to approximately 3.1% and 2.6% for precision and recall while still performing computations on a GPU. Each scenario will have different priorities, but this work can be used as a guide for configuring a training environment with respect to nondeterminism and training time.

Acknowledgements. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), supported by NSF grant DMR200035. We thank Prof. Danielle Cote and Bryer Sousa for providing the dataset and insights.

References

1. Nvidia docs (2018). <https://github.com/NVIDIA/framework-determinism>
2. Identifying, evaluating, and addressing nondeterminism in mask R-CNNs source code (2021). <https://github.com/Data-Driven-Materials-Science/Nondeterminism>
3. Amza, C., et al.: Treadmarks: shared memory computing on networks of workstations. *Computer* **29**(2), 18–28 (1996)
4. Bengio, Y., Grandvalet, Y.: No unbiased estimator of the variance of k-fold cross-validation. *J. Mach. Learn. Res.* **5**, 1089–1105 (2004)
5. Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010, pp. 177–186. Springer (2010). https://doi.org/10.1007/978-3-7908-2604-3_16
6. Bottou, L.: Stochastic gradient descent tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *Neural Networks: Tricks of the Trade*. LNCS, vol. 7700, pp. 421–436. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35289-8_25
7. Brown, S.T.E.A.: Bridges-2: a platform for rapidly-evolving and data intensive research. *Pract. Exp. Adv. Res. Comput.* (2021)
8. Buck, I.: GPU computing with Nvidia Cuda. In: *ACM SIGGRAPH 2007 Courses*, pp. 6-es (2007)

9. Chen, Z., Wang, J., He, H.: A fast deep learning system using GPU. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE (2014)
10. Collaboration, O.S., et al.: Estimating the reproducibility of psychological science. *Science* **349**, 6251 (2015)
11. Defour, D., Collange, S.: Reproducible floating-point atomic addition in data-parallel environment. In: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 721–728. IEEE (2015)
12. Documentation, C.T.: v10. 2. Nvidia Corp., Santa Clara (2019)
13. Garland, M.E.A.: Parallel computing experiences with Cuda. *IEEE Micro* **28**(4), 13–27 (2008)
14. Gill, J.: Computational complexity of probabilistic turing machines. *SIAM J. Comput.* **6**(4), 675–695 (1977)
15. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **23**(1), 5–48 (1991)
16. He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask R-CNN. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2961–2969 (2017)
17. He, Z., Rakin, A.S., Fan, D.: Parametric noise injection: trainable randomness to improve deep neural network robustness against adversarial attack. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 588–597 (2019)
18. Kahan, W.: IEEE standard 754 for binary floating-point arithmetic. *Lect. Notes Status IEEE* **754**(94720–1776), 11 (1996)
19. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simulat.* **8**(1), 3–30 (1998)
20. Nagarajan, P., Warnell, G., Stone, P.: The impact of nondeterminism on reproducibility in deep reinforcement learning (2018)
21. Paszke, A.E.A.: Pytorch: an imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.* **32**, 8026–8037 (2019)
22. Price, S.E., Gleason, M.A., Sousa, B.C., Cote, D.L., Neamtu, R.: Automated and refined application of convolutional neural network modeling to metallic powder particle satellite detection. In: Integrating Materials and Manufacturing Innovation, pp. 1–16 (2021)
23. Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. *J. Big Data* **6**(1), 1–48 (2019)
24. Shuryak, I.: Advantages of synthetic noise and machine learning for analyzing radioecological data sets. *PloS One* **12**(1) (2017)
25. Steinkraus, D., Buck, I., Simard, P.: Using GPUS for machine learning algorithms. In: Eighth International Conference on Document Analysis and Recognition (ICDAR 2005), pp. 1115–1120. IEEE (2005)
26. VanRossum, G., Drake, F.L.: The Python Language Reference. Python Software Foundation, Amsterdam (2010)
27. Wang, Y.E., Wei, G.Y., Brooks, D.: Benchmarking TPU, GPU, and CPU platforms for deep learning. arXiv preprint [arXiv:1907.10701](https://arxiv.org/abs/1907.10701) (2019)
28. Wu, Y., Kirillov, A., Massa, F., Lo, W.Y., Girshick, R.: Detectron2 (2019). <https://github.com/facebookresearch/detectron2>
29. Zhong, Z., Zheng, L., Kang, G., Li, S., Yang, Y.: Random erasing data augmentation. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 13001–13008 (2020)