machine learning & knowledge extraction

MDPI

Article

How Do Deep-Learning Framework Versions Affect the Reproducibility of Neural Network Models?

Mostafa Shahriari *, Rudolf Ramler and Lukas Fischer

Software Competence Center Hagenberg GmbH (SCCH), 4232 Hagenberg, Austria
* Correspondence: m.shahriari.sh@gmail.com

Abstract: In the last decade, industry's demand for deep learning (DL) has increased due to its high performance in complex scenarios. Due to the DL method's complexity, experts and non-experts rely on blackbox software packages such as Tensorflow and Pytorch. The frameworks are constantly improving, and new versions are released frequently. As a natural process in software development, the released versions contain improvements/changes in the methods and their implementation. Moreover, versions may be bug-polluted, leading to the model performance decreasing or stopping the model from working. The aforementioned changes in implementation can lead to variance in obtained results. This work investigates the effect of implementation changes in different major releases of these frameworks on the model performance. We perform our study using a variety of standard datasets. Our study shows that users should consider that changing the framework version can affect the model performance. Moreover, they should consider the possibility of a bug-polluted version before starting to debug source code that had an excellent performance before a version change. This also shows the importance of using virtual environments, such as Docker, when delivering a software product to clients.

Keywords: machine learning; deep learning; deep neural networks; reproducibility; tensorflow; pytorch

Citation: Shahriari, M.; Ramler, R.; Fischer, L. How Do Deep-Learning Framework Versions Affect the Reproducibility of Neural Network Models? *Mach. Learn. Knowl. Extr.* **2022**, *4*, 888–911. https://doi.org/10.3390/make4040045

Academic Editor: Weiping Ding

Received: 18 August 2022
Accepted: 30 September 2022
Published: 5 October 2022

1. Introduction

In the last decade, deep-learning (DL) algorithms have been increasing daily due to their efficiency in solving highly complicated problems [1]. Recently, we can find a trace of deep neural networks (DNNs) in many applications, such as computer vision [2], natural language processing and speech recognition [3], biometrics [4], and geophysics [5,6], to mention a few. Before training, a DNN is a parametric representation of the function governing the desired process. Then, by minimizing a loss function using some stochastic processes, such as stochastic optimization, we fit the DNN to a specific dataset. Hence, given input from the dataset, a DNN can produce output with generality [7]. Nonetheless, nondeterminism is a commonly known phenomenon in engineering ML/DL systems [8–10].

The daily advances in DL and its complex low-level implementations compelled giant technology companies such as Google and Meta AI to invest in creating open-source high-level DL packages.

The most common DL framework is Tensorflow, developed and maintained by Google (Mountain View, CA, USA). They mention on their website: "TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources. It lets researchers push the state-of-the-art in ML, and developers easily build and deploy ML-powered applications." [11]. The first version of Tensorflow was released in 2015. The current updated versions of Tensorflow were released under the name *Tensorflow* 2.0 whose first version was released in 2019.

Another well-known framework with increasing popularity, especially for academic users, is Pytorch. Pytorch is developed and maintained by Meta AI (New York, NY, USA): "An open source machine learning framework that accelerates the path from research

prototyping to production deployment." [12]. The first version of Pytorch was released in 2016. Table 1 briefly compares the two above-mentioned frameworks.

Table 1. A feature comparison between Tensorflow and Pytorch.

	Tensorflow	**Pytorch**
Creator	Google	Meta (Facebook)
Computational graph creation	static/dynamic	dynamic
API level	high and low level	high and low level
Debugging	difficult	easy
Ease of use	Incomperehensive API	Integrated with Python

Nowadays, the increase in the acquired data volume and the improvements in hardware components are leading to the popularity of DL methods. Therefore, the reliability of a proposed DL technique is of paramount importance. The reproducibility of DNN models implemented using DL frameworks is critical in showing their reliability. However, mainly, the following reasons interfere with the reproducibility of models using these frameworks:

- Randomization in DL training methods: A DNN training and optimization process includes a high level of randomization, e.g., weight initialization and random batch selection, and stochastic optimization, e.g., stochastic gradient descent [13–17]. Indeed, this problem arises using many machine-learning (ML) methods [18–21]. It is possible to reduce randomization by deactivating some functionalities, e.g., randomized batch selection at each epoch; however, this may also decrease the model performance.
- GPU implementations: The DL frameworks use Cuda [22] and cudnn [23] for their GPU implementations to accelerate DNN training. These libraries introduce randomization in their implementations to expedite processes, e.g., selecting primitive operations, floating point precision, and matrix operations [24,25].
- Bugs in DL frameworks: DL frameworks are software, after all. As with any software, bugs can be introduced in the development process of DL frameworks. This problem can be magnified when a new version of the framework contains features that fail or show deteriorating performance although working correctly in previous versions [26–32].
- Improvement in methods and implementations: As DL is an active field of research, it faces continuous and rapid improvements. Hence, responsible developers implement state-of-the-art advancements daily to keep the frameworks updated. These improvements can also lead to changes in the output of the DL-based codes.

Researchers using DL frameworks should know the aforementioned irreproducibility issues to produce reliable methods and results. The authors of [16] performed a survey by asking more than 900 researchers and developers to fill out a detailed questionnaire. Surprisingly, many researchers and developers were unaware of these problems or their severity. This shows that many researchers use DL frameworks as a blackbox without awareness of the processes and potential pitfalls. This problem becomes even more severe for users of software systems developed on top of DL frameworks, as they may be completely unaware of the issues introduced by using frameworks in the first place. This emphasizes the importance of users being aware of basic DL/ML principles and receiving training and post-installation support from researchers and developers. Moreover, researchers and developers should provide monitoring and reporting mechanisms as part of their software solutions which provide insights into the underlying processes and the performance of the applied models.

This work investigates reproducibility issues related to DNNs when using different versions of DL frameworks and their effects using quantitative measures. We focus on the two most common DL frameworks: Tensorflow and Pytorch. We use well-known problems and simple DNNs to show the variance in the model's performance obtained using different framework versions. To restrict our study, we only perform the training on the CPU

to reduce the level of uncertainty arising from GPU-related implementations. Hence, comparing two separate versions, we can obtain different variances in the performance of the obtained models mainly because of code changes and related bugs introduced in the DL frameworks during the development process. The main aim of this study is to bring awareness to researchers and developers using DL and about the problems they may face when they upgrade/downgrade to another version of the DL framework in use. Table 2 summarizes the objectives of this work. Finally, we propose solutions for users and developers to control and monitor training to achieve the best performance in their final DNN model.

Table 2. List of objectives to address in this work that affect the performance of a DL-based code or its resulting model's performance.

RQ1	influence of different DL framework versions
RQ2	noticeable effect of bugs in DL framework versions
RQ3	differences between two popular frameworks

The remainder of this work is organized as follows: Section 2 describes the study we perform in this work, i.e., the DL definition, the investigated use cases, DNN architecture design, and the training process. Section 3 shows the results of our experiments and analyses them. Section 4 is dedicated to the conclusion and discussion.

2. Study Design

In this study, we explore the impact of version changes in different DL frameworks on the reproducibility of trained models. Table 2 describes the investigated objectives of this work. We consider well-known use cases/datasets to study the aforementioned objectives. For each use case, we define a simple DNN architecture. Ultimately, by training the DNNs using different versions of the DL frameworks and comparing their results, we investigate the effect of version change in the DL-based code and the resulting model performance.

2.1. DL Definition

We consider $X = \{x_0, x_1, \cdots, x_n\}$ and $Y = \{y_0, y_1, \cdots, y_n\}$ to be the input and output spaces, respectively, and we have $f(x_i) = y_i$ for all $(x_i, y_i) \in X \times Y$, where f is the target function. In the context of supervised DL, knowing the input and output spaces, we aim to approximate the target function f using a DNN. The parametric representation of the aforementioned DNN approximation $f_{\mathbf{w},\mathbf{b}}$ is as follows:

$$f_{\mathbf{w},\mathbf{b}} = f_{w_n,b_n} o f_{w_{n-1},b_{n-1}} o \cdots o f_{w_0,b_0}, \tag{1}$$

where f_{w_i,b_i} is the function governing the i-th layer of the DNN with w_i and b_i being its weights and bias matrices, respectively; o shows the function composition, $\mathbf{w} = \{w_0, w_1, \cdots, w_n\}$, and $\mathbf{b} = \{b_0, b_1, \cdots, b_n\}$. Then, the training of the DNN minimizes the following loss function:

$$\mathbf{w}^*, \mathbf{b}^* = \arg\min_{\mathbf{w},\mathbf{b}} \mathcal{L}(f_{\mathbf{w},\mathbf{b}}(X), Y), \tag{2}$$

where $\mathcal{L}$ is a loss function comparing the DNN prediction and the ground truth in a predefined regime. Then, the DNN approximation of f is $f_{\mathbf{w}^*,\mathbf{b}^*}$.

In this work, we consider classification problems. Hence, the output space Y is a set of valid classes/categories.

2.2. Investigated Cases

For our study, we selected well-known and widely used classification problems with corresponding datasets as use cases: Pulsars, Iris species, heart disease, 2D Gaussian

distribution, and body mass index (BMI). Table 3 describes all the datasets and their features.

Table 3. List of datasets used as use cases for our experiments.

Dataset	List of Features
Pulsars	mean of the integrated profile, the standard deviation of the integrated profile, excess kurtosis of the integrated profile, skewness of the integrated profile, mean of the dispersion measure signal to noise ratio (DM-SNR) curve, the standard deviation of the DM-SNR curve, excess kurtosis of the DM-SNR curve, skewness of the DM-SNR curve.
Iris Species	sepal length (cm), sepal width (cm), petal length (cm), petal width (cm).
Heart Disease	age, sex, chest pain type (four values), resting blood pressure, serum cholesterol in ($\frac{mg}{dl}$), resting electrocardiographic results (three values), maximum heart rate achieved, exercise-induced angina, ST depression induced by exercise relative to rest, the slope of the peak exercise ST segment, number of major vessels (0–3) colored by fluoroscopy, Thal(0 = normal, 1 = fixed defect, 2 = reversible defect).
2D Gaussian distribution	points location along x- and y- axis
Body Mass Index	gender, height (cm), weight (kg)

2.2.1. Pulsars Dataset

A *Pulsar* is a neutron star that produces a detectable radio emission on Earth. Each sample in this dataset consists of eight continuous variables and one class. The class is a Boolean variable. This dataset contains 17,898 samples in which 1639 are positive, and the rest are negative (https://www.kaggle.com/datasets/colearninglounge/predicting-pulsar-starintermediate (accessed on 1 August 2022)).

We first remove the samples containing a missing value in the preprocessing part. Then, we rescale all the inputs to be between 0 and 1. Then, as the dataset is unbalanced, we use the synthetic minority over-sampling technique (SMOTE) to balance the dataset. We fix the random seed for the SMOTE to produce the same samples consistently [33].

2.2.2. Iris Species Dataset

This dataset is dedicated to classifying the species of the Iris plant using its flower characterizations. It contains three separate classes of species. Hence, the problem is a multi-class prediction. This dataset includes 150 samples equally divided between three target iris species, i.e., 50 samples for each class (https://www.kaggle.com/datasets/uciml/iris (accessed on 1 August 2022)).

For preprocessing, we first rescale all the input values to $(0, 1)$. Then, we apply a one-hot encoding to the label values.

2.2.3. Heart Disease Dataset

In this dataset, we use a patient's information to predict if they have any heart disease. The output is Boolean, detecting the presence of heart disease. This dataset contains $1,025$ samples (https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset (accessed on 1 August 2022)).

We only rescale the input values to $[0, 1]$ in preprocessing.

2.2.4. Two-Dimensional (2D) Gaussian Distribution

We describe a joint Gaussian distribution of a 2D random vector $X = (x_0, x_1)$ as $X \sim \mathcal{N}(\boldsymbol{\mu}, \sigma)$. In the aforementioned notation, $\boldsymbol{\mu} = (\mu_0, \mu_1)$, where μ_i is the mean of the random variable x_i. Moreover, as we consider a joint distribution, we have $\sigma = \begin{bmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{bmatrix}$, where σ_i is the standard deviation of x_i. In this experiment, we consider two separate 2D Gaussian distributions, i.e., $X^1 \sim \mathcal{N}(\boldsymbol{\mu}^1, \sigma^1)$ and $X^2 \sim \mathcal{N}(\boldsymbol{\mu}^2, \sigma^2)$. Therefore, given a vector of random variables X_p, the classification task consists of predicting $j \in \{1, 2\}$ where $X_p \sim \mathcal{N}(\boldsymbol{\mu}^j, \sigma^j)$.

Using 2D Gaussian distribution, we produce a dataset of 5000 samples. The input in this dataset is a vector of random variables, and the output is the class showing its corresponding Gaussian distribution. In this experiment, we consider $\boldsymbol{\mu}^1 = [0, 10]$ and $\boldsymbol{\mu}^2 = [10, 0]$. Moreover, we consider $\sigma^1 = \sigma^2 = 10 * I_2$, where I_2 is the identity matrix of dimension 2×2. Figure 1 shows the samples.

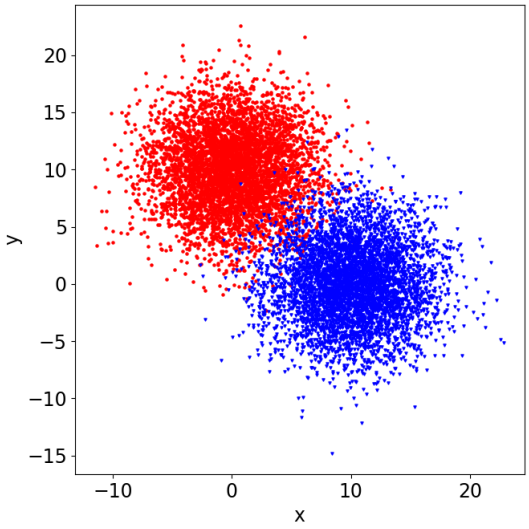

Figure 1. Visualization of the Gaussian points belong to two different distributions. The blue points show $X^1 \sim \mathcal{N}(\boldsymbol{\mu}^1, \sigma^1)$, and the red points represent $X^2 \sim \mathcal{N}(\boldsymbol{\mu}^2, \sigma^2)$.

In the preprocessing stage, we rescale the input to be in the $[0, 1]$ interval.

2.2.5. BMI Dataset

In this dataset, using the physical information of participants, we predict the BMI index. The BMI index helps us to detect obesity. The corresponding index is an integer value between one to five. Hence, the output is five separate classes. This dataset contains 500 samples (https://www.kaggle.com/code/titan23/bmi-dataset/notebook (accessed on 1 August 2022)).

In the preprocessing stage, we first encode the gender values to unique binary values. Then, we rescale all the input to $[0, 1]$.

2.3. DNN Architecture

There exist types of DNN layers that consist of randomization and stochastic operations, e.g., the pooling layer. To avoid the aforementioned randomization, we only consider fully connected layers to produce our DNN architectures. Table 4 describes the DNN architectures used for all the model problems. In all DNNs, except the final layer, a ReLu activation function follows the output of each fully connected layer. For the final layer, in the case of binary classification, the activation function is Sigmoid. In the case of multi-class classification, the last layer contains a Softmax activation function.

Table 4. The NN architecture used for each model problem. We only use fully connected layers.

Model Problem	Layers	Optimizer	Epochs
Pulsars	$(64, 64, 1)$	Adam	10
Iris species	$(256, 64, 32, 32, 3)$	Rmsprop	100
Heart disease	$(128, 64, 1)$	Adam	75
2D Gaussian	$(10, 10, 1)$	Adam	10
BMI	$(64, 64, 6)$	Adam	100

In the case of Tensorflow, we consider the following versions (and their corresponding release dates): 2.2.0 (May 2020), 2.2.3 (June 2021), 2.3.0 (July 2020), 2.3.4 (August 2021), 2.4.0 (December 2020), 2.4.4 (December 2021), 2.5.0 (May 2021), 2.5.3 (February 2022), 2.6.2 (November 2021), 2.6.3 (February 2022), 2.7.0 (November 2021), 2.7.1 (February 2022), 2.8.0 (February 2022), 2.9.0-*rc*2 (May 2022). Usually, Tensorflow developers release a 2.x.0 version that includes major improvements compared to the previous versions concerning efficient computational capabilities and recent developments in the field. Then, versions 2.x.y are dedicated to fixing reported bugs and deficiencies of 2.x.0 versions.

For Pytorch, we use the following versions (and their corresponding release dates): 1.6.0 (July 2020), 1.7.0 (October 2020), 1.7.1 (December 2020), 1.8.0 (March 2021), 1.8.1 (March 2021), 1.9.0 (June 2021), 1.9.1 (September 2021), 1.10.0 (October 2021), 1.11.0 (March 2022), 1.12.0 (June 2022). Pytorch is following an analogous release strategy to Tensorflow.

2.4. Experimental Setting

To eliminate the randomization from the Cuda and cudnn implementations when using a GPU setup [22,23], we perform the experiments on the CPU. We use a standard desktop PC with 2 GHz Quad-Core Intel Core *i*5 as the processor for this task.

As the training process includes levels of randomization, e.g., initialization, and stochastic processes, e.g., optimization, to perform a fair comparison among the different versions of the DL platforms, we run the experiments multiple times for each version. By performing various experiments, we conclude that it is sufficient to run them twenty times to obtain the variance for the model performance. However, this selection is arbitrary, and one might consider fewer or more repetitions. Therefore, we obtain a variance in the models' performance for each version. Before training, we split our dataset into training and validation datasets, i.e., 80% and 20% of the entire dataset, respectively. We consider a fixed random seed to obtain similar training and validation datasets in each repetition. After training, we save the resulting model. Then, by loading the model and evaluating it on the validation dataset, we obtain inference results. Hence, to compare the results of each version, we produce the following figures:

- Initial accuracy: The model's accuracy on the validation dataset before training. The optimal initialization of the DNN is an active line of research and encounters continuous improvements. These improvements also affect the development of the frameworks. The proper initialization of the DNN is of paramount importance as different initial values can lead the optimizer to separate local minimums, hence, different results.
- Final training accuracy: The model's accuracy on the validation dataset at the last epoch. This value shows us what to expect if the user trains the model using the same training setup.
- Best accuracy: The best accuracy of the model on the validation dataset during training. We may encounter the best performance of the model before the final epoch. This quantity magnifies the importance of training with care and supervision. Furthermore, it is one of the reasons to save the checkpoints while training that the user can retrieve the best model after the training.
- Epoch with the best accuracy: the identifier of the epoch that the best performance of the model was achieved.

- Inference accuracy: The accuracy of the final model (the resulting model after the last epoch) at the inference stage. We measure this value to identify any bug in the DL framework.
- Average final accuracy: The average of the last models' accuracies in twenty runs for each version. To obtain reproducible results, it is common practice to take an average of the outputs of multiple runs. This value illustrates what to expect if we use this approach.

3. Results

This section presents the results for the investigated DL frameworks, Tensorflow and Pytorch.

3.1. Tensorflow Models

In this framework, we faced a compatibility error/bug when using Tensorflow version 2.6.0 (https://stackoverflow.com/questions/72255562/cannot-import-name-dtensor-from-tensorflow-compat-v2-experimental (accessed on 10 August 2022)). This sort of behavior causes users frustration and confusion when upgrading the systems. It makes a working application crash with no fault of the application itself. For the rest of the versions, we summarize the results as follows:

3.1.1. Pulsars

Figure 2 shows the results of this experiment. For the initial accuracy, excluding the outliers, versions 2.5.3 and 2.9.0-*rc*2 show the maximum variation (almost 2%). However, there are evident differences among different versions. Different initial points can lead the optimizer to different local minimums of the loss function. These variations can lead to inconsistent solutions when training the same model using separate versions of Tensorflow. Analogously, in the case of final accuracy, the maximum variance happens in versions 2.3.0 and 2.4.0, i.e., almost 1%. The maximum accuracy while training also shows discrepancies between different versions. However, by checking the average accuracy for all versions, we see that this difference is not significant enough to impose any problem at the inference stage if we use an averaging approach. The epoch in which the maximum accuracy happens shows total randomness caused by many factors, e.g., the initialization. However, similar to initial accuracy, versions 2.5.3 and 2.9.0-*rc*2 show the maximum variance among the considered versions. We expect the initial point selection to lead to faster or slower convergence to the local minimum. Hence, a high variance in initial accuracy can lead to a high variance in the epoch with maximum accuracy when the optimizer parameters are constant.

3.1.2. Iris Species

Figure 3 describes the results of this experiment. For the initial accuracy, we witness randomness. However, the average initial accuracy for all the versions is almost similar. In the case of final accuracy, we notice an extreme variation (more than 10%) in some versions, e.g., 2.2.3 and 2.5.3. This discrepancy can result in an inapplicable model in the production stage. In all versions, in a specific epoch, the model achieves its best performance (100% accuracy). However, this accuracy can happen in any epoch, and different versions also affect the epoch identifier. The average accuracy between separate models is also different, i.e., it can deliver inapplicable models. Version 2.5.3 shows the minimum average accuracy of almost 2.5% lower than 2.8.0.

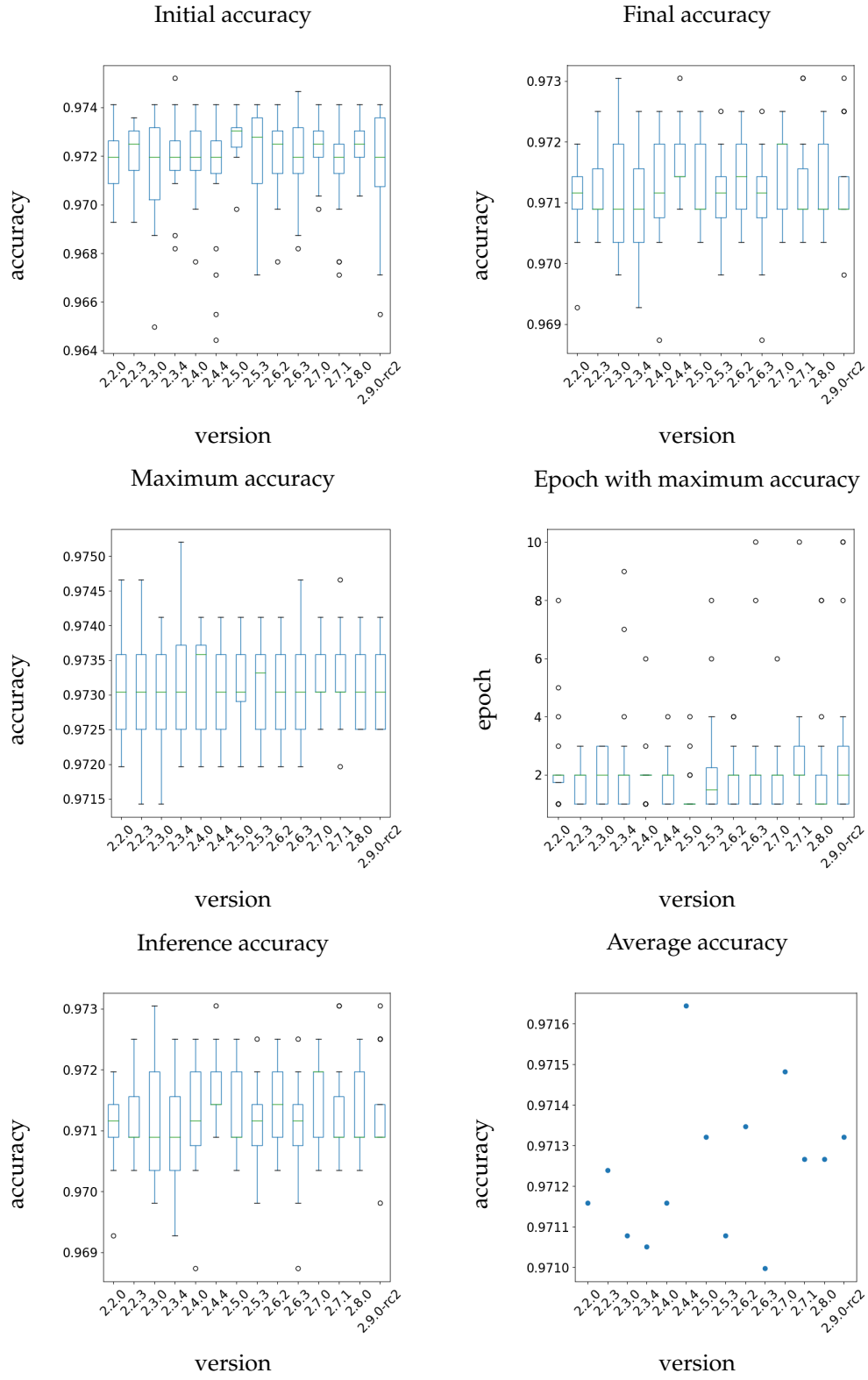

Figure 2. The results for the Pulsars dataset using Tensorflow.

3.1.3. Heart Disease

Figure 4 presents the results for this experiment. These results also show similar conclusions to the Iris experiment. In contrast to Pulsars, version 2.5.3 and 2.9.0-*rc*2 show low variance in cases of initial accuracy. This shows that, considering the nature of the problem and the Tensorflow functions that we need to use for each problem, the efficiency

of the DL-based code and its resulting model can improve or deteriorate when using different versions of Tensorflow.

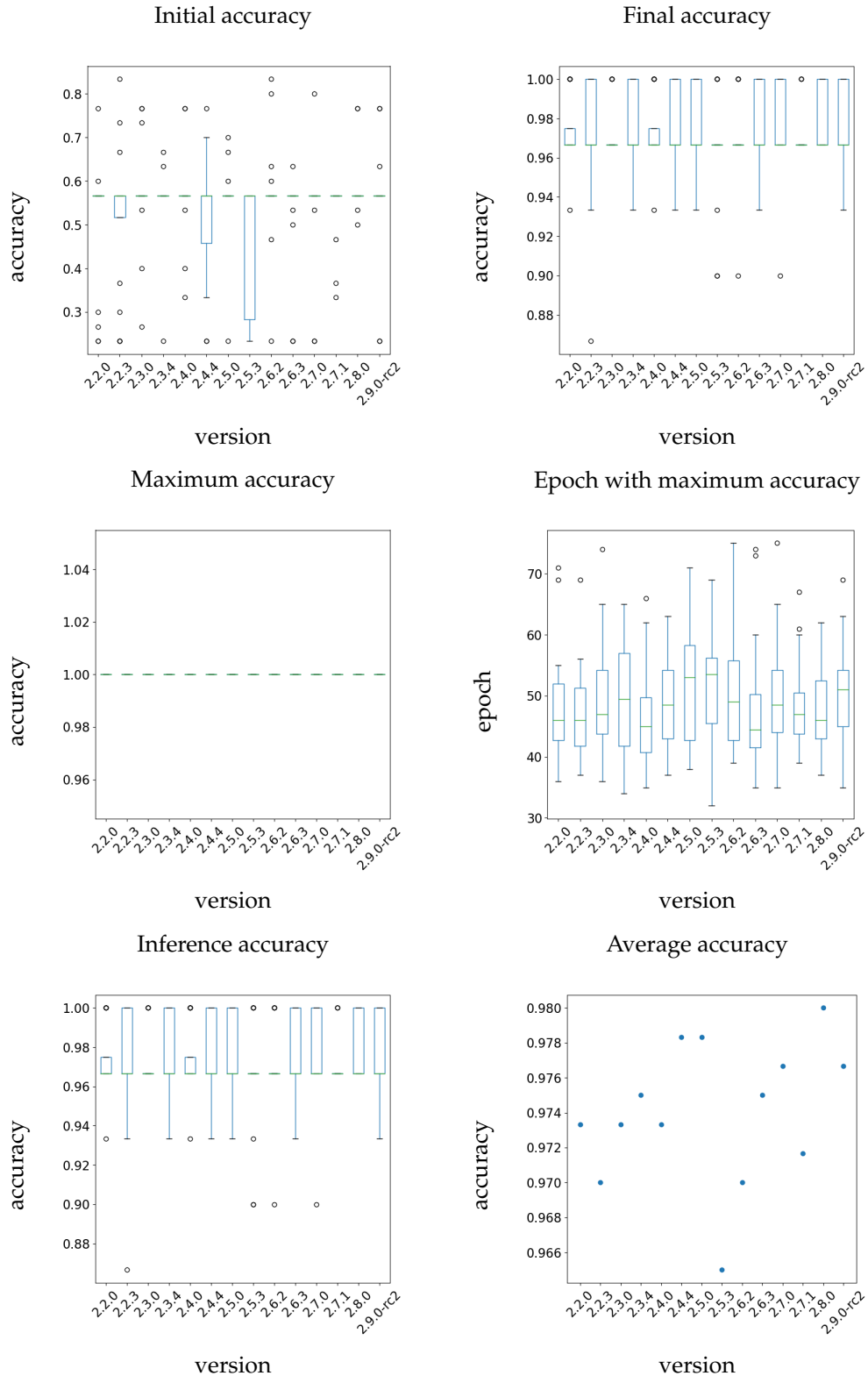

Figure 3. The results for the Iris dataset using Tensorflow.

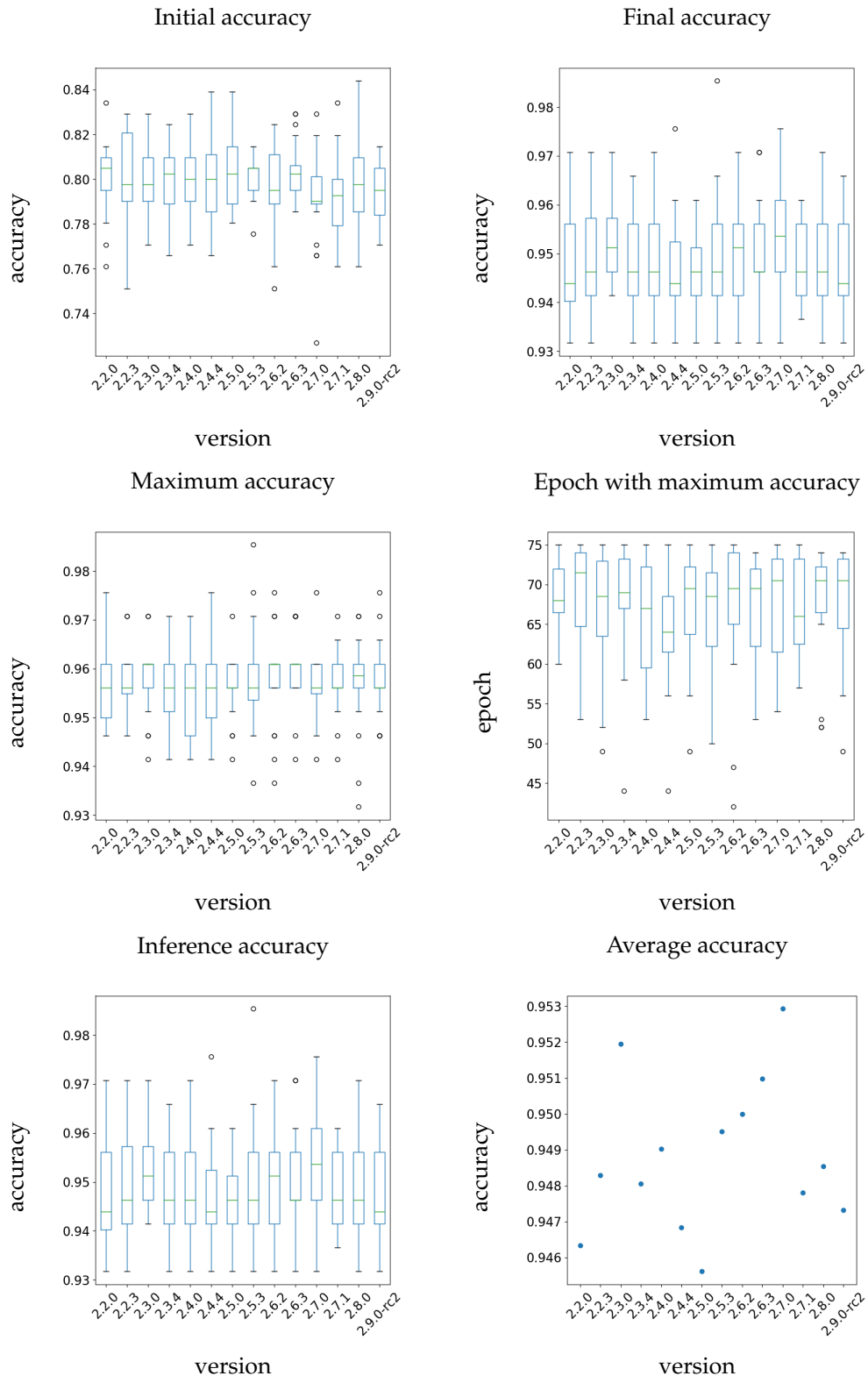

Figure 4. The results for the heart-disease dataset using Tensorflow.

3.1.4. 2*D* Gaussian

Figure 5 presents the results for this experiment. These results also show similar conclusions to the previous experiments. Similarly, we encounter the minimum average accuracy at version 2.5.3, which is almost 2% lower than the maximum in version 2.2.0.

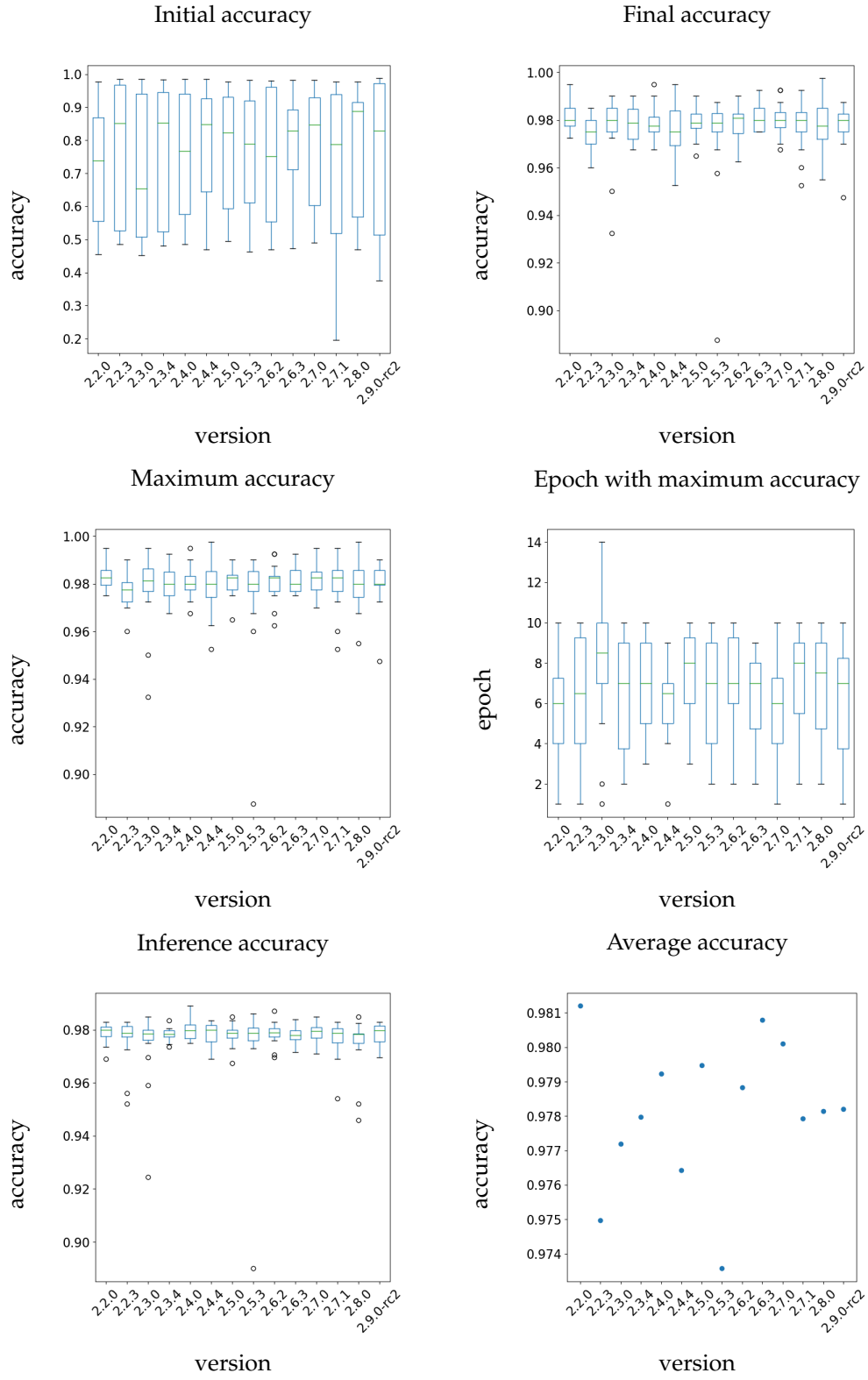

Figure 5. The results for the 2D Gaussian dataset using Tensorflow.

3.1.5. BMI

Figure 6 shows the results of this experiment. The results show similar discrepancies comparing the different versions of Tensorflow. However, we can also see a failure in evaluating the model using versions 2.3.0 and 2.3.4 that can be related to a documented bug in this version (https://github.com/tensorflow/tensorflow/issues/42459 (accessed on 10

August 2022)). This bug does not affect the prediction. However, the evaluation results are unreliable, leading to confusion and, more importantly, to wrong and unstable results.

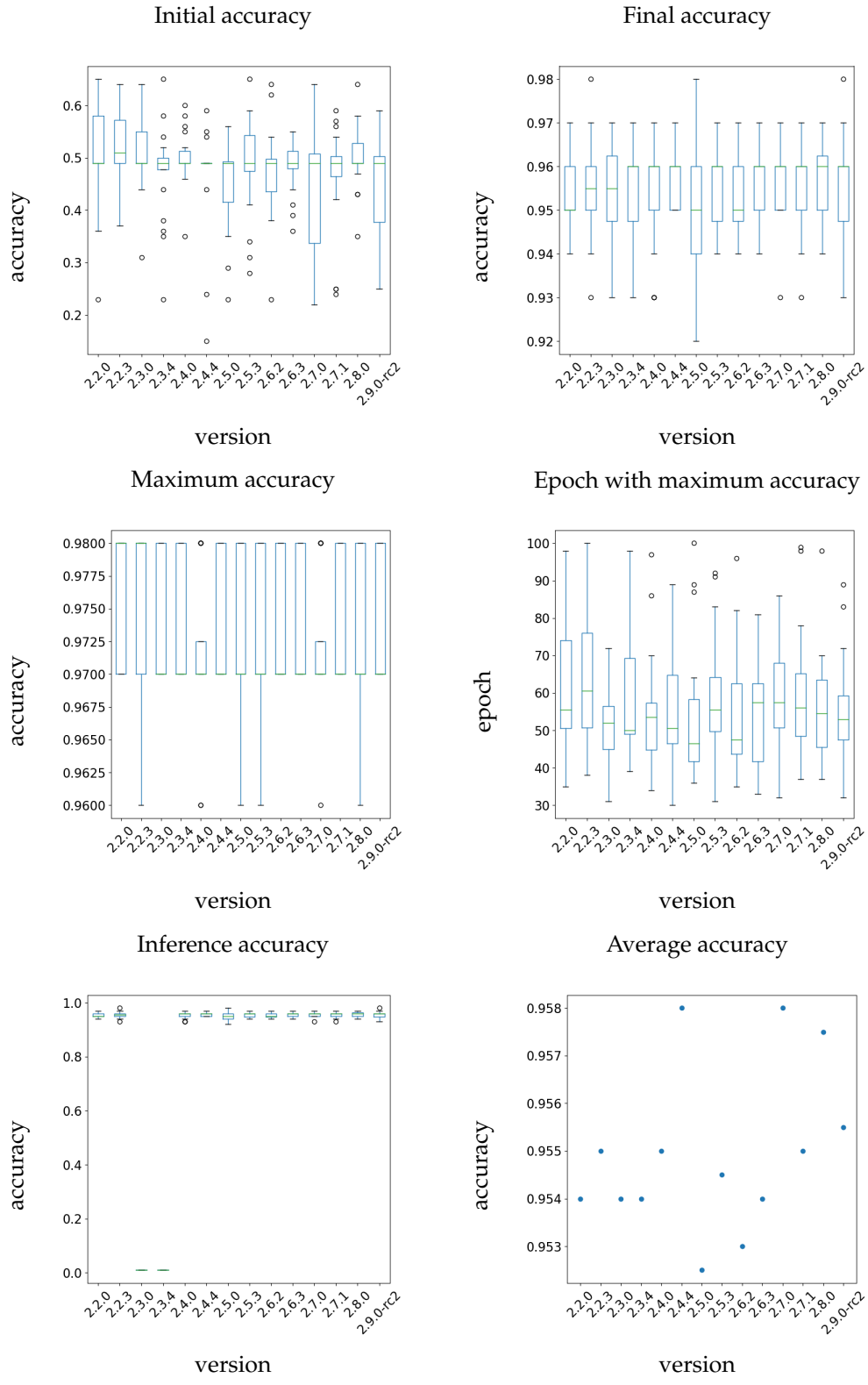

Figure 6. The results for the BMI dataset using Tensorflow.

3.2. Pytorch Models

In the case of Pytorch models, we conclude our results as follows:

3.2.1. Pulsars

Figure 7 shows the results for this experiment. The difference between the initial accuracy of the networks in different versions is evident. Versions 1.9.1 and 1.7.0 show the maximum and minimum variation, respectively. The final accuracy of the DNNs shows almost similar behavior in different versions. The randomization can cause slight differences in this case. Maximum and average training accuracies are also similar in all versions. The variation is too small to cause any inconvenience in inference, i.e., producing inapplicable models. Considering the number of epochs for which we achieve the maximum accuracy, we witness some discrepancies that show the importance of monitoring the training stage.

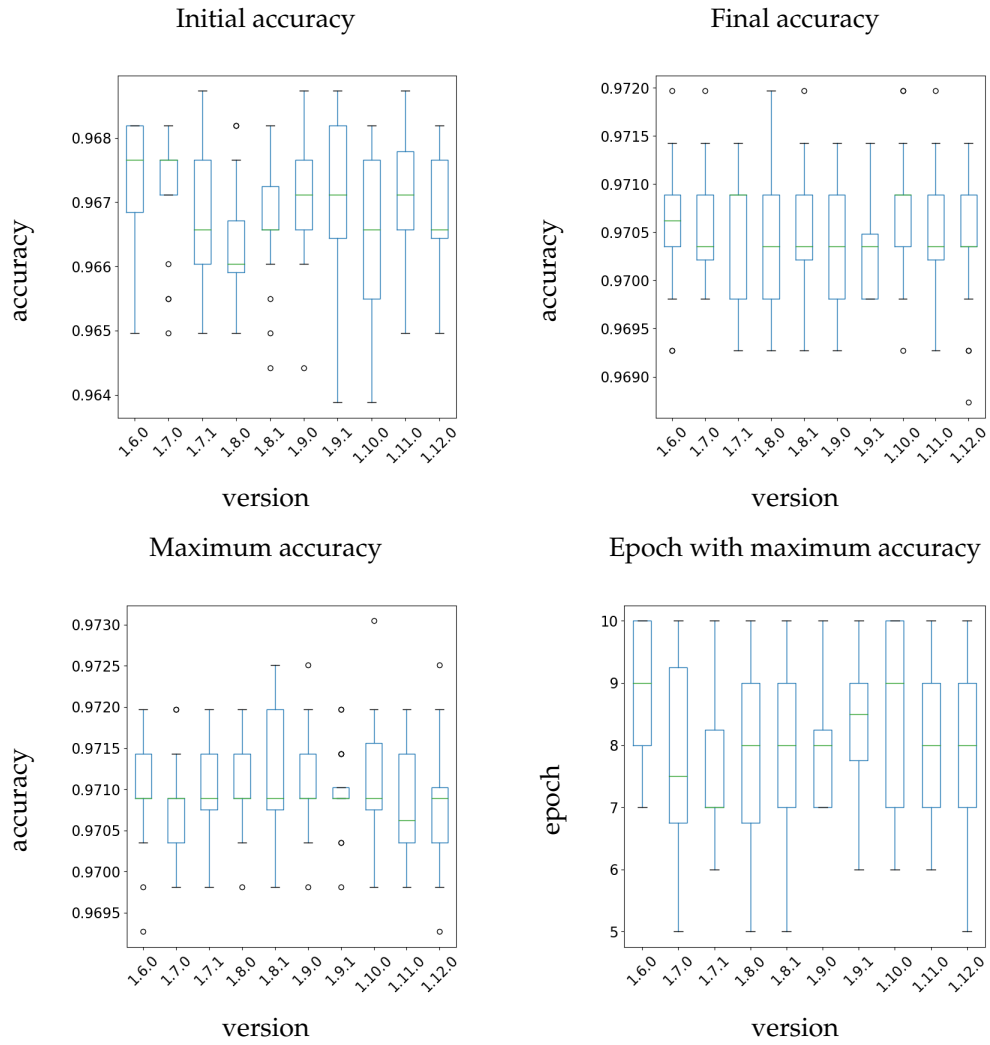

Figure 7. *Cont.*

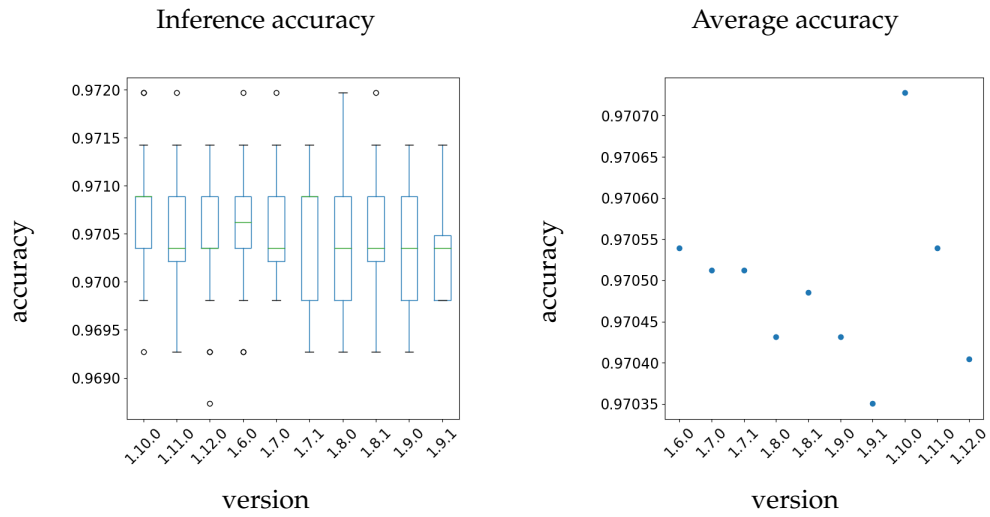

Figure 7. The results for the Pulsar dataset using Pytorch.

3.2.2. Iris Species

Figure 8 represents the results of this experiment. The final and average accuracies show a significant discrepancy among different versions. This discrepancy can be up to 1.5%, which can cause unreliability in the model. We see a significant difference among versions considering the epoch number at which we reach the maximum accuracy. Excluding the outliers, versions 1.7.0, 1.8.0, and their corresponding subversions 1.7.1 and 1.8.1 show the maximum variance. In this case, version 1.6.0 shows the minimum variance.

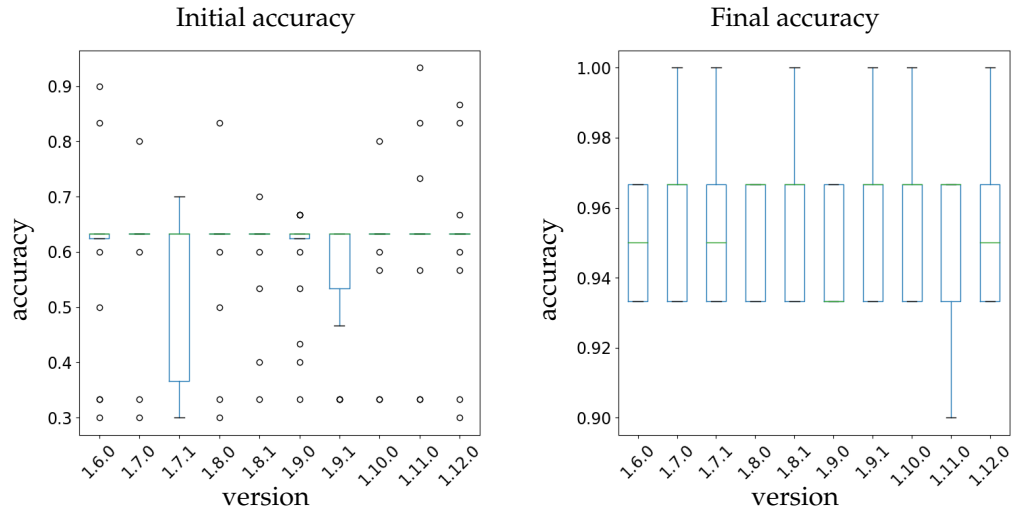

Figure 8. *Cont.*

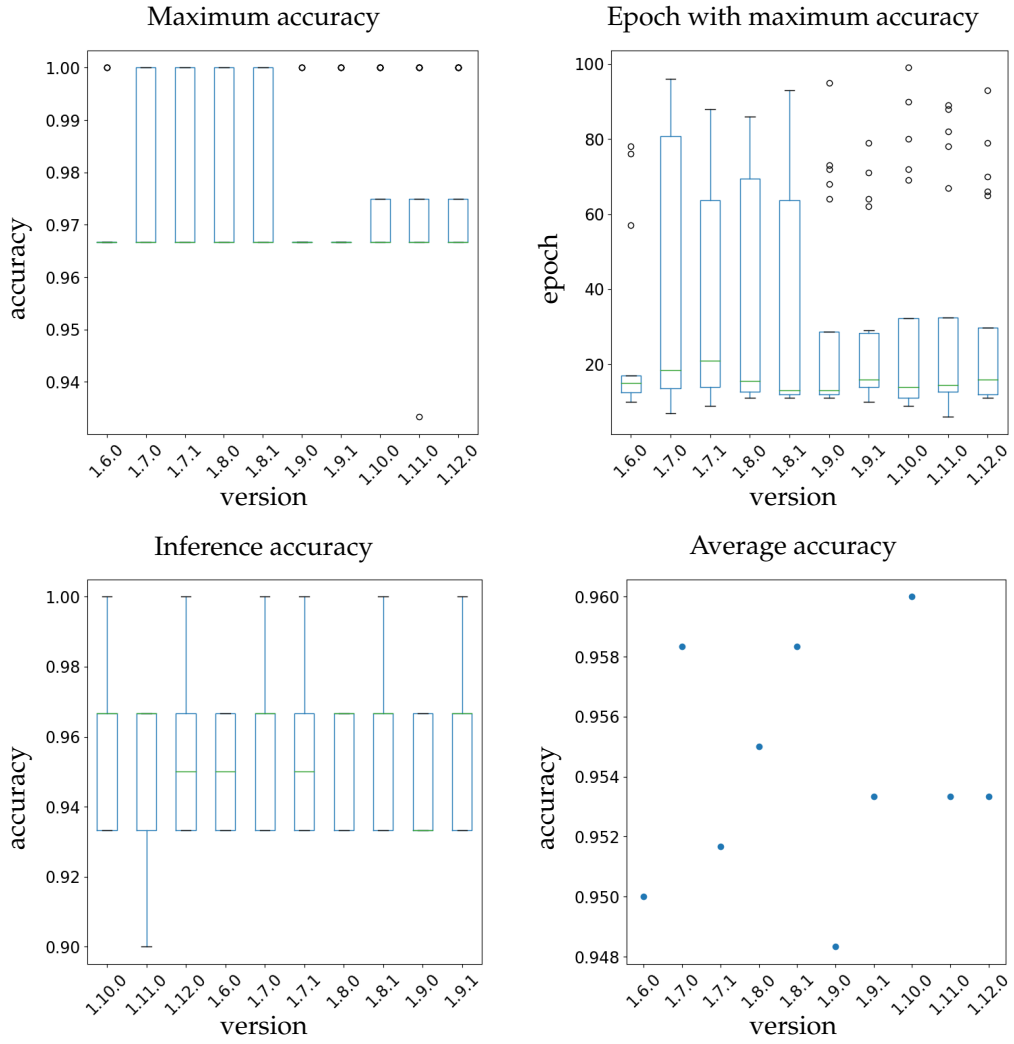

Figure 8. The results for the Iris dataset using Pytorch.

3.2.3. Heart Disease

Figure 9 shows the results. For the final accuracy, a significant difference exists among different runs in one version that can make the models unreliable. Moreover, some versions, e.g., 1.10.1, achieve better accuracy than others. The average accuracy also verifies this difference. The number of epochs with maximum accuracy also shows discrepancies among versions.

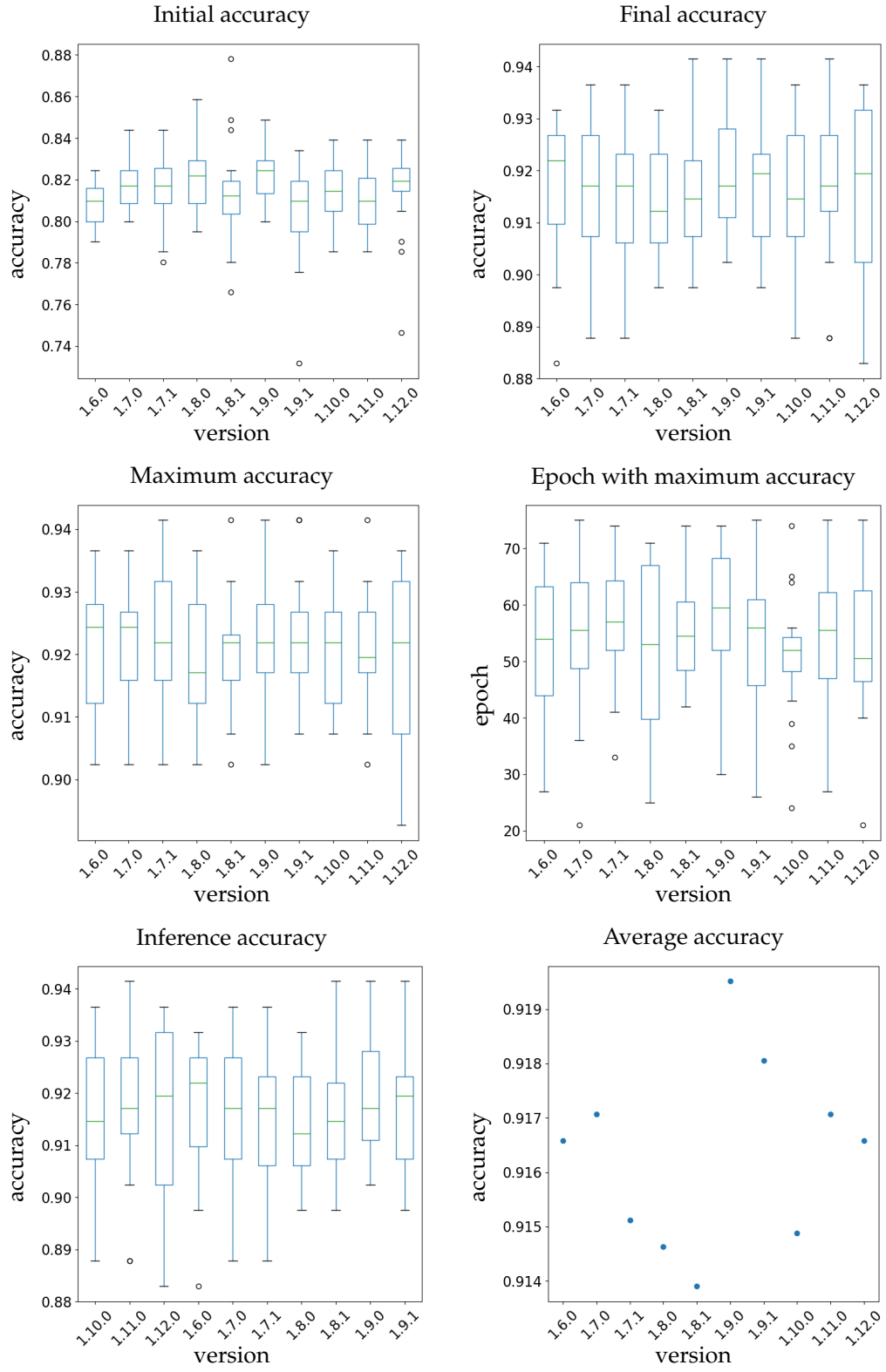

Figure 9. The results for the heart-disease dataset using Pytorch.

3.2.4. 2*D* Gaussian

Figure 10 shows the results of this experiment. In this case, the maximum, final, and average accuracies show concerning results in version 1.8.0.

3.2.5. BMI

Figure 11 shows this experiment's results. In this case, the epoch with maximum accuracy shows a significant variance among different versions.

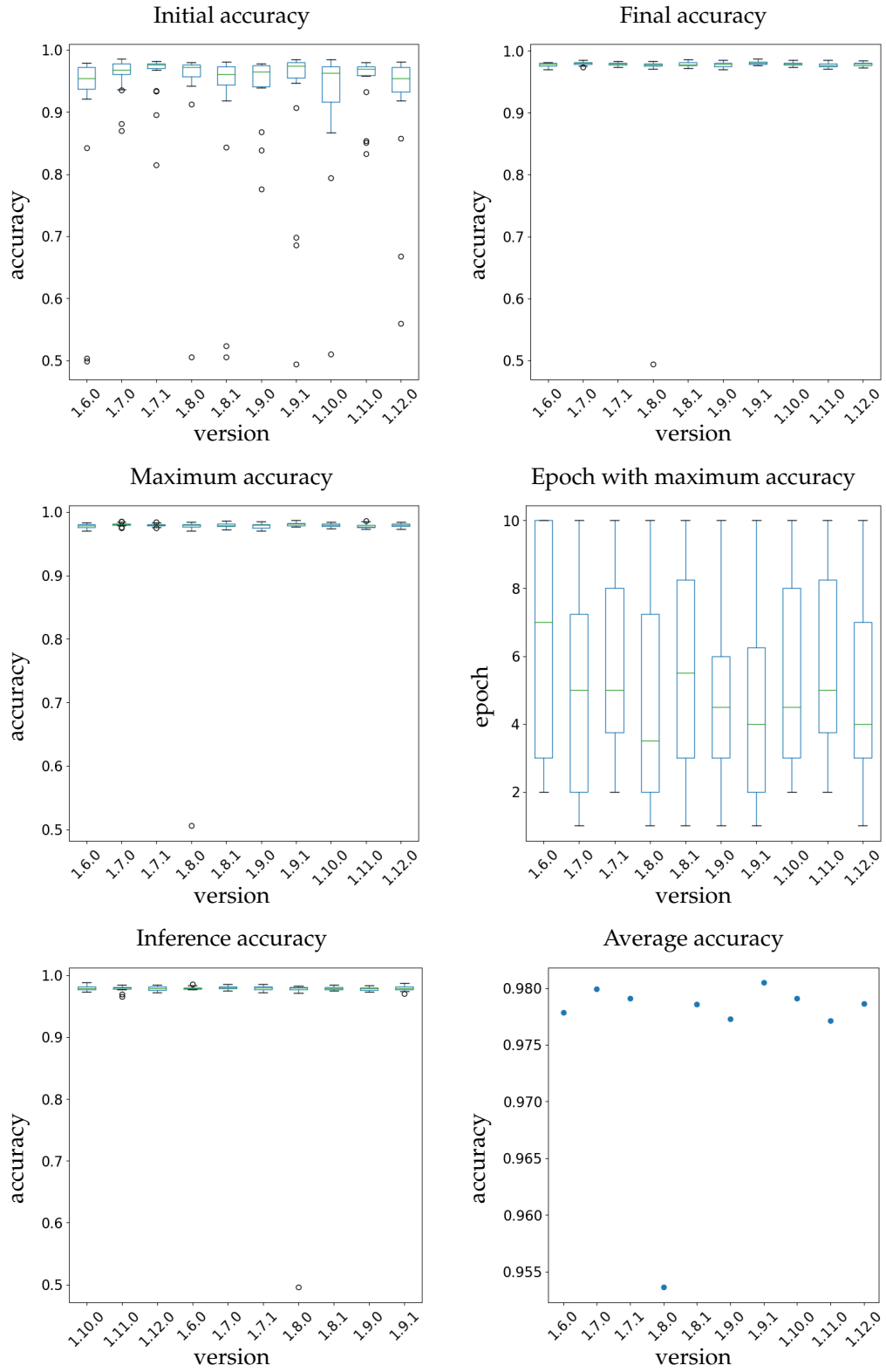

Figure 10. The results for the 2D Gaussian dataset using Pytorch.

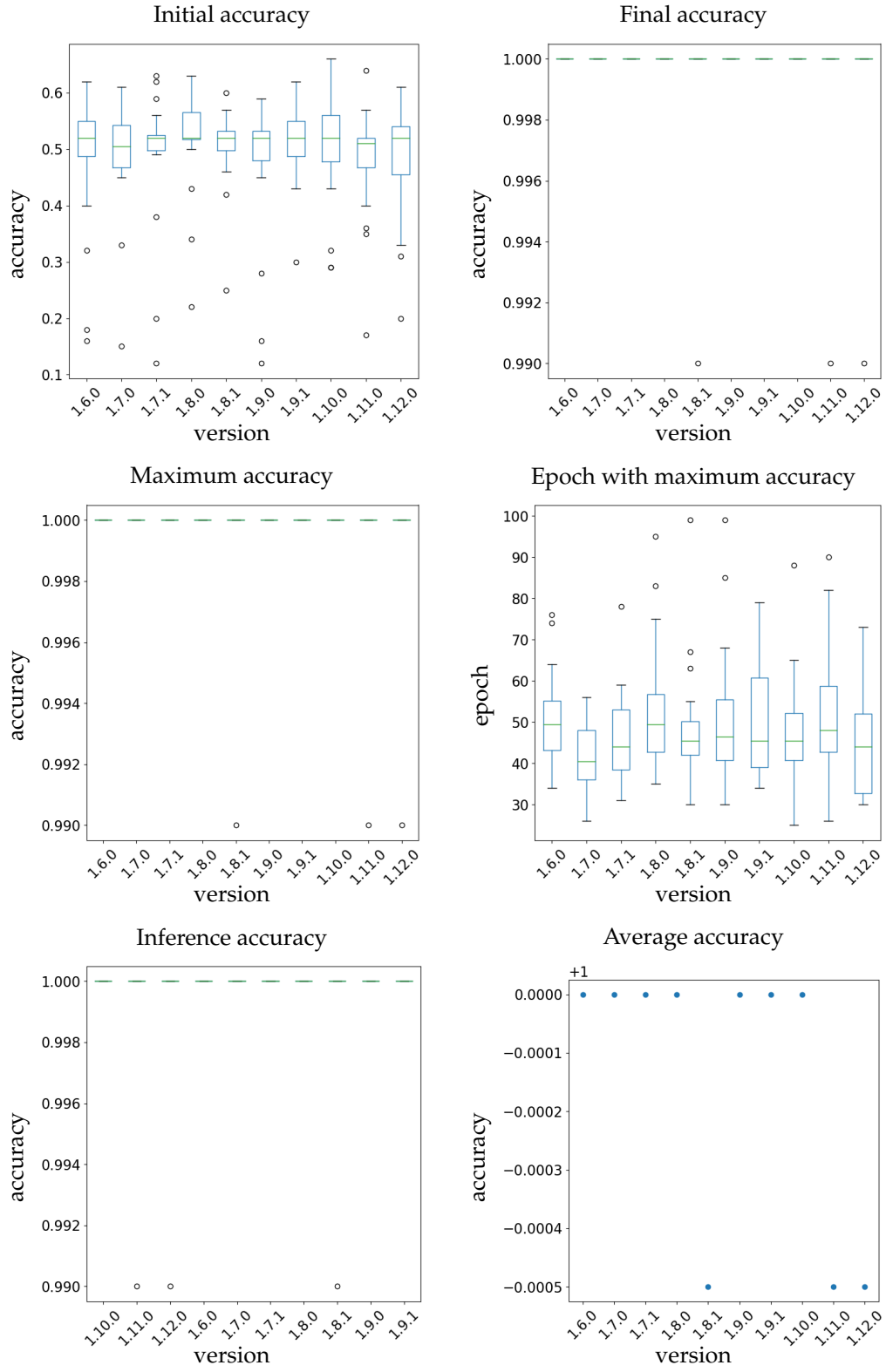

Figure 11. The results for the BMI dataset using Pytorch.

4. Discussion

4.1. Answer to Research Questions

Regarding the influence of version change on the reproducibility of the DNN model, we respond to the objectives raised in Table 2 as follows:

- RQ1: This irreproducibility can occur when using different versions of these frameworks. Due to the implementation changes, any upgrade/downgrade of these frameworks can increase/reduce the variance in the model performance. The aforementioned changes can also occur in the dependencies of these frameworks. Nevertheless, any of these changes leads to a change in the variance of the model's performance.
- RQ2: A bug introduced by the developer may lead to erroneous results or the DL code to crash. In our use cases, we faced this issue in Tensorflow versions 2.6.0, 2.3.0, and 2.3.4.
- RQ3: In our considered cases, by comparing the results between Tensorflow and Pytorch, we find no reason to conclude that either of them can produce more reliable models in the context of reproducibility when the framework version changes. However, in Pytorch, we did not witness any bug that caused a sudden performance deterioration or code crash.

4.2. Comparison to Related Work

The focus of our work is on the impact of DL framework versions on model performance. Research related to our work has been conducted with the focus on (a) repeatability and reproducibility of DL results, (b) bugs in ML/DL frameworks and components, and (c) software engineering best practices for DL. In these areas, several studies have also identified nondeterministic effects as a critical factor for producing reliable and repeatable results with DL, yet from a different perspective and on a less-detailed level. In the following, we discuss the related work and differences to our study.

4.2.1. Repeatability and Reproducibility

Repeatability is the ability to obtain the exact same results of an experiment under the same experimental setup, such as hardware and software settings on multiple runs. It is the precondition for reproducing an experiment to obtain the same results by an independent team following documented procedures. The importance of reproducibility when using DL is rapidly increasing due to more and more sensitive and safety-critical data-science applications in recent years [34].

However, repeatability issues are frequent in DL [15] and, in consequence, DL is facing a serious reproducibility challenge [35,36] which is gaining more and more attention in the research community.

Alahmari et al. [15] studied repeatability issues in training DL models with two frameworks (Pytorch and Keras) using the same data under the same software and hardware settings. They showed that even when applying the available control of randomization in Keras and TensorFlow, there are uncontrolled randomizations due to variations in the implementation of the weight initialization algorithm across deep-learning libraries. However, in contrast to our work, they did not evaluate the impact on repeatability caused by operating systems and deep-learning framework versions.

Zhuang et al. [37] conducted a series of experiments across different types of hardware, accelerators, state-of-the-art networks, and open-source datasets, to assess the impact of tooling choices on the level of non-determinism in DL. They found that both algorithmic and implementation noise have a significant impact. Implementation noise includes noise introduced by the selected DL framework (e.g., Tensorflow, PyTorch, cuDNN) as well as hardware acceleration architectures (e.g., CPU/GPU). They did not specifically analyze the impact of different software versions of the selected DL frameworks.

In a recent study, Gundersen et al. [21] conducted a comprehensive literature review on the sources of irreproducibility. They identified six groups of influence factors: (1) study design factors, (2) algorithmic factors, (3) implementation factors, (4) observation factors, (5) evaluation factors, and (6) documentation factors. Implementation factors affecting reproducibility comprise different initialization seeds but also the same seed on different platforms, truncation errors of floating point calculations with single precision (32 bits) or double-precision (64 bits), parallel executions leading to a random completion order of parallel tasks, changing processing units such as switching from CPU to GPU and vice versa,

the use of different DL frameworks such as TensorFlow or PyTorch, different operating systems, as well as different software versions of involved libraries, DL frameworks or operating systems. While they identify different software versions as a relevant influence factor, their work does not provide a quantification of the related influence.

Qian et al. [38] quantified the impact of the variance introduced by DL software implementations. They found that identical DL training runs (i.e., identical network, data, configuration, software, and hardware) with a fixed seed produce different models with a large variance in fairness, up to 12.6%. Hence, one training run may produce a fair model but another fixed-seed identical training run may generate an unfair one. In their work, the impact of variance is quantified, but not at the level of individual influence factors.

4.2.2. Bugs in DL Software

DL frameworks are widely used by non-experts. However, like any other programs, they are prone to bugs. These bugs can lead to, e.g., crashes, bad performance, incorrect output, data corruption, or memory leakage [39]. Bad performance refers to the consequence of a bug where the accuracy of the trained model is negatively affected. The severity of such bugs is particularly high if these bugs occur "silently", i.e., without the user noticing it [27].

Bugs can occur in DL frameworks, in programs written by users, or in the data. According to Islam et al. [32], data bugs and logic bugs are the most severe bug types in deep-learning software. In their study, they examined several hundred posts from Stack Overflow and bug fix commits from Github about five popular deep-learning libraries Caffe, Keras, Tensorflow, Theano, and Torch. They also identified fast changes in new DL framework versions as a major challenge. For example, they report that almost 26% of operations were changed from version 1.10 to 2.0 in TensorFlow.

Jia et al. [40] analyzed 202 bugs inside the TensorFlow framework, which they collected directly from closed pull requests on GitHub. They identified the following bug categories: Functional errors (35.6%), where the software does not function as expected; crash (26.7%), when the software aborts unexpectedly; hang (1.5%), when the software keeps running without responding; performance degradation (1.5%), when the software does not provide results in expected time; build failure (23.8%), when the software cannot be compiled in the first place; and warning-style error (10.9%), when warning messages are shown in the build process.

The subcategory of bugs named "silent bugs" has been studied by Tambon et al. [27]. These bugs lead to the wrong behavior of the system, but they do not cause crashes or hangs, nor do they indicate any error message to the user. Such bugs are even more dangerous in DL applications and frameworks due to the black-box and non-deterministic nature of the systems, which makes it hard for the end user to understand the model and explain decisions. Tambon et al. found 77 reproducible silent bugs in TensorFlow and Keras from their respective GitHub repositories. They identified several categories of effect caused by silent bugs: the wrong shape of a tensor in the model without raising an error, wrong/deceiving information displayed on the user interface or console, wrong or incomplete saving/reloading of the model, wrong parameter setting, degrading runtime or memory performance, wrong model structure, and wrong calculations resulting in incorrectly computed results.

In these categories, bugs of type "wrong calculation" (e.g., back-propagation gradients being computed wrongly) and "wrong saving/reloading" (e.g., weights not being properly set when a saved model is reloaded) have the highest severity as these bugs represent issues that would drastically affect the results of the model without obvious noticeable symptoms for the user. The authors advise not blindly trusting DL frameworks as they are not infallible, and results should always be carefully and critically reviewed and compared to similar studies or a baseline.

In this context, our work complements the findings from these studies, and it describes an approach for revealing silent regression bugs by comparing training results from consecutive versions of DL frameworks.

4.2.3. Software Engineering Best Practices

Amershi et al. [41] report on a study that has been conducted on observing software teams at Microsoft developing AI-based applications, providing insights about several essential engineering challenges that organizations may face in creating large-scale AI solutions. They identified three main challenges in AI engineering that make it fundamentally different to software engineering: (1) provisioning and managing data for DL applications is much more complex than for developing software applications, (2) model customization and model reuse require new skills not typically found in software teams, and (3) ML/DL models are more difficult to handle as they are entangled in complex ways, and because they exhibit non-deterministic behavior.

The authors describe several best practices for applying ML/DL in software engineering, including, for example, building end-to-end pipeline support to automate model training, deployment, and integration with the product they are a part of. Furthermore, they also elaborate on best practices for model evolution, evaluation, and deployment since ML/DL applications go through frequent revisions initiated by model tuning, data changes, and software updates, which have a significant impact on system performance. Frequent model iterations also require frequent deployment, which should be accompanied by automated tests that ensure that models work as intended after every update.

A systematic literature review on the state of software engineering research for engineering ML/DL systems conducted by Giray [42] identified similar practices. In particular, the author emphasized the challenges arising due to the non-deterministic nature of ML/DL on all engineering aspects of ML/DL systems. Testing has been identified as one of the far most popular measures to address these issues in the reviewed research.

Overall, while most of the related works recognize and discuss the non-deterministic behavior of ML/DL as an important source of issues when developing ML/DL systems, the analysis of these issues, their effects, as well as the underlying causes are studied on a very abstract and broad level.

Nevertheless, there exists one study in context of engineering DL software systems, by Pham et al. [16], that specifically examines the variance in DL systems and the factors that introduce nondeterminism. The authors quantitatively analyze the variance related to model accuracies and training times resulting from factors introducing nondeterminism over multiple identical training runs (e.g., identical training data, algorithm, and network). Besides algorithmic factors, DL frameworks and libraries (e.g., TensorFlow and cuDNN) introduce additional variance referred to as implementation-level variance due to parallelism, optimization, and floating-point computation. These implementation-level factors alone cause an accuracy difference across identical training runs of up to 2.9%, a per-class accuracy difference of up to 52.4%, and a training time difference of up to 145.3%.

All investigated DL frameworks (TensorFlow, CNTK, and Theano) and DL libraries (e.g., cuDNN) also exhibit implementation-level variance across different versions. In this study, the authors also analyzed the overall accuracy differences of 11 low-level library combinations (cuDNN and CUDA) with TensorFlow to examine the variance when switching versions of the low-level libraries. They observed an average overall accuracy difference of 2% (largest overall accuracy difference of 2.9% and smallest 1.6%) in fixed-seed identical training runs with the 11 library combinations. With respect to the analysis of different version combinations, the study conducted by Pham et al. is closely related to our work. In our study, we were able to identify cases exhibiting even larger differences in accuracy, which are confirmed by the findings described in [16] .

5. Conclusions

In this work, we investigated the effect of version change on model performance in two common DL frameworks, Tensorflow and Pytorch. We selected a set of well-known datasets/examples to compare the performance of the aforementioned DL frameworks. For each use case, we designed a simple DNN consisting of multiple fully connected layers. We utilized only fully connected layers to reduce the level of stochastic processes that can

arise from the nature of the DNN layer, e.g., pooling layers. Moreover, as the problems' computational complexity is low, we train the models using a CPU to avoid randomization caused by Cuda and cudnn implementations. Using a GPU implementation can only increase the level of uncertainty that we witness. Using the aforementioned experimental setup, we analyzed the performance of different stable versions of the frameworks to obtain a quantitative analysis of their corresponding models' performance.

The results of a single version show that the randomization involved in the DNN training, e.g., initialization and optimization, hinders the reproducibility of the obtained model. In some cases, e.g., Pulsars, this variation is negligible. However, we may obtain unacceptable results in other cases, e.g., Iris. This variation can be the difference between an efficient and an inapplicable model. Moreover, when considering two separate DNN models, i.e., two models with different architectures, a slight improvement in the results corresponding to one of the models compared to the other is not enough to judge its superiority.

At the academic level, the variations mentioned above in the model's performance can throw into question the reliability of some research work. Moreover, if a model is used in an industrial setting, an upgrade or a downgrade in the framework's version or its dependencies can reduce the performance of an already installed model and may lead to catastrophic consequences. As randomization increases when using GPUs for training, it can only magnify the abovementioned problems. To control the training process and to reduce the issues arising from the reproducibility of the DNN models, we suggest the following:

- To use virtual environments, such as Docker, to deliver a model to any industrial partner. Using these environments saves the model from any version change during an upgrade.
- To use graphics, such as the ones we used, to properly investigate the model's efficiency before using it in any industrial cycle.
- To save the checkpoints while training the model. By doing so, the user can use a model with a better performance obtained in the previous epochs.
- To avoid using DL codes as a blackbox. As we witnessed, in the best-case scenario, the best-performing model can be in an earlier epoch than the one defined for the training. The user should be able to control and adapt these variables to achieve maximum efficiency.
- To use automated-ML frameworks, e.g., KerasTuner, to obtain the model with the best performance [43]. Using these techniques, we can extract the model with the best performance by defining a search space of variables, e.g., learning rate. In advance usage, we can use these techniques to select the best model architecture in a designated search space of DNN architectures.
- To avoid using the output of a single training as the sole evaluator of the model performance. In academic works, one can claim with caution that a model performs better than others as the model performance can change if we repeat the training process.

In future work, we will investigate the irreproducibility caused by changing the hardware components, i.e., CPU and GPU. Moreover, we shall study this effect in less common DL frameworks, e.g., Caffe. We shall also examine the impact of DNN layers that contain stochastic processes, e.g., the pooling layer, on the model's performance.

Author Contributions: Conceptualization, M.S., R.R. and L.F.; methodology, M.S., R.R. and L.F.; software, M.S.; validation, M.S., R.R. and L.F.; formal analysis, M.S.; investigation, M.S. and R.R.; resources, R.R. and L.F.; data curation, M.S.; writing—original draft preparation, M.S.; writing—review and editing, M.S., R.R. and L.F.; visualization, M.S.; supervision, R.R. and L.F.; project administration, R.R.; funding acquisition, R.R. and L.F. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Austrian Ministry for Transport, Innovation, and Technology (BMVIT), the Federal Ministry for Digital and Economic Affairs (BMDW), the Province of Upper Austria in the frame of the COMET-Competence Centers for Excellent Technologies Program managed by Austrian Research Promotion Agency FFG, and FFG Bridge project Contest Nr. 888127.

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: All datasets used in this research are publicly available. For further details see Section 2.2. Pulsars dataset: https://www.kaggle.com/datasets/colearninglounge/predicting-pulsar-starintermediate; Iris Species dataset: https://www.kaggle.com/datasets/uciml/iris; Heart Disease dataset: https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset; BMI dataset https://www.kaggle.com/code/titan23/bmi-dataset/notebook.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fischer, L.; Ehrlinger, L.; Geist, V.; Ramler, R.; Sobiezky, F.; Zellinger, W.; Brunner, D.; Kumar, M.; Moser, B. AI System Engineering—Key Challenges and Lessons Learned. *Mach. Learn. Knowl. Extr.* **2021**, *3*, 56–83. [CrossRef]
2. Lu, L.; Zheng, Y.; Carneiro, G.; Yang, L. *Deep Learning for Computer Vision: Expert Techniques to Train Advanced Neural Networks Using TensorFlow and Keras*; Springer: Cham, Switzerland, 2017.
3. Yu, D.; Deng, L. *Automatic Speech Recognition: A Deep Learning Approach*; Springer: London, UK, 2017.
4. Bhanu, B.; Kumar, A. *Deep Learning for Biometrics*; Springer: Cham, Switzerland, 2017.
5. Shahriari, M.; Hazra, A.; Pardo, D. A deep learning approach to design a borehole instrument for geosteering. *Geophysics* **2022**, *87*, D83–D90. [CrossRef]
6. Shahriari, M.; Pardo, D.; Rivera, J.A.; Torres-Verdín, C.; Picón, A.; Ser, J.D.; Ossand'on, S.; Calo, V.M. Error control and loss functions for the deep learning inversion of borehole resistivity measurements. *Int. J. Numer. Methods Eng.* **2021**, *122*, 1629–1657. [CrossRef]
7. Higham, C.F.; Higham, D.J. Deep learning: An introduction for applied mathematicians. *Comput. Res. Repos.* **2018**, *61*, 860–891. [CrossRef]
8. Eiben, A.; Smith, J. *Introduction to Evolutionary Computing*; Springer: Berlin/Heidelberg, Germany, 2015. [CrossRef]
9. Debnath, P.; Mohiuddine, S. *Soft Computing Techniques in Engineering, Health, Mathematical and Social Sciences*; CRC Press: Boca Raton, FL, USA, 2021.
10. Debnath, P.; Castillo, O.; Kumam, P. *Soft Computing: Recent Advances and Applications in Engineering and Mathematical Sciences*; CRC Press: Boca Raton, FL, USA, 2023.
11. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://www.tensorflow.org/ (accessed on 1 August 2022).
12. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc.: Red Hook, NY, USA, 2019; pp. 8024–8035.
13. Somepalli, G.; Fowl, L.; Bansal, A.; Yeh-Chiang, P.; Dar, Y.; Baraniuk, R.; Goldblum, M.; Goldstein, T. Can Neural Nets Learn the Same Model Twice? Investigating Reproducibility and Double Descent from the Decision Boundary Perspective. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, New Orleans, LA, USA, 18–24 June 2022; Available online: http://xxx.lanl.gov/abs/2203.08124 (accessed on 1 August 2022).
14. Nagarajan, P.; Warnell, G.; Stone, P. The Impact of Nondeterminism on Reproducibility in Deep Reinforcement Learning. 2018. Available online: https://openreview.net/forum?id=S1e-OsZ4e7 (accessed on 1 August 2022).
15. Alahmari, S.S.; Goldgof, D.B.; Mouton, P.R.; Hall, L.O. Challenges for the Repeatability of Deep Learning Models. *IEEE Access* **2020**, *8*, 211860–211868. [CrossRef]
16. Pham, H.V.; Qian, S.; Wang, J.; Lutellier, T.; Rosenthal, J.; Tan, L.; Yu, Y.; Nagappan, N. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 December 2020; Association for Computing Machinery: New York, NY, USA, 2020; ASE '20, pp. 771–783.
17. Hartley, M.; Olsson, T.S. dtoolAI: Reproducibility for Deep Learning. *Patterns* **2020**, *1*, 100073. [CrossRef]
18. Crane, M. Questionable Answers in Question Answering Research: Reproducibility and Variability of Published Results. *Trans. Assoc. Comput. Linguist.* **2018**, *6*, 241–252. [CrossRef]
19. Beam, A.; Manrai, A.; Ghassemi, M. Challenges to the Reproducibility of Machine Learning Models in Health Care. *JAMA* **2020**, *323*. [CrossRef]
20. Gundersen, O.E.; Shamsaliei, S.; Isdahl, R.J. Do machine learning platforms provide out-of-the-box reproducibility? *Future Gener. Comput. Syst.* **2022**, *126*, 34–47. [CrossRef]

21. Gundersen, O.E.; Coakley, K.; Kirkpatrick, C. Sources of Irreproducibility in Machine Learning: A Review. *arXiv* **2022**, arXiv:2204.0761.
22. NVIDIA; Vingelmann, P.; Fitzek, F.H.P. CUDA, Release: 10.2.89. 2020. Available online: https://developer.nvidia.com/cuda-toolkit (accessed on 1 August 2022).
23. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. cuDNN: Efficient Primitives for Deep Learning. *arXiv* **2014**, arXiv:1410.0759. Available online: http://xxx.lanl.gov/abs/1410.0759 (accessed on 1 August 2022).
24. Struski, L.; Morkisz, P.; Spurek, P.; Bernabeu, S.R.; Trzcinski, T. Efficient GPU implementation of randomized SVD and its applications. *arXiv* **2021**, arXiv:2110.03423. Available online: http://xxx.lanl.gov/abs/2110.03423 (accessed on 1 August 2022).
25. Liberty, E.; Woolfe, F.; Martinsson, P.G.; Rokhlin, V.; Tygert, M. Randomized algorithms for the low-rank approximation of matrices. *Proc. Natl. Acad. Sci. USA* **2008**, *104*, 20167–20172. [CrossRef] [PubMed]
26. Rivera-Landos, E.; Khomh, F.; Nikanjam, A. The Challenge of Reproducible ML: An Empirical Study on The Impact of Bugs. In Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 6–10 December 2021; pp. 1079–1088. [CrossRef]
27. Tambon, F.; Nikanjam, A.; An, L.; Khomh, F.; Antoniol, G. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. *arXiv* **2021**, arXiv:2112.13314.
28. Leotta, M.; Olianas, D.; Ricca, F.; Noceti, N. How Do Implementation Bugs Affect the Results of Machine Learning Algorithms? In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; Association for Computing Machinery: New York, NY, USA, 2019; SAC '19, pp. 1304–1313. [CrossRef]
29. Zhang, Y.; Chen, Y.; Cheung, S.C.; Xiong, Y.; Zhang, L. An Empirical Study on TensorFlow Program Bugs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; Association for Computing Machinery: New York, NY, USA, 2018; ISSTA 2018; pp. 129–140. [CrossRef]
30. Dwarakanath, A.; Ahuja, M.; Sikand, S.; Rao, R.M.; Bose, R.P.J.C.; Dubash, N.; Podder, S. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; Association for Computing Machinery: New York, NY, USA, 2018; ISSTA 2018; pp. 118–128. [CrossRef]
31. Humbatova, N.; Jahangirova, G.; Bavota, G.; Riccio, V.; Stocco, A.; Tonella, P. Taxonomy of Real Faults in Deep Learning Systems. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea, 5–11 October 2020; Association for Computing Machinery: New York, NY, USA, 2020; ICSE '20, pp. 1110–1121. [CrossRef]
32. Islam, M.J.; Nguyen, G.; Pan, R.; Rajan, H. A Comprehensive Study on Deep Learning Bug Characteristics. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; ESEC/FSE 2019, pp. 510–520. [CrossRef]
33. Chawla, N.; Bowyer, K.; Hall, L.; Kegelmeyer, W. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res. (JAIR)* **2002**, *16*, 321–357. [CrossRef]
34. Gundersen, O.E.; Gil, Y.; Aha, D.W. On reproducible AI: Towards reproducible research, open science, and digital scholarship in AI publications. *AI Mag.* **2018**, *39*, 56–68. [CrossRef]
35. Gundersen, O.E.; Kjensmo, S. State of the art: Reproducibility in artificial intelligence. In Proceedings of the AAAI Conference on Artificial Intelligence, Orleans, France, 2–7 February 2018; Volume 32.
36. Haibe-Kains, B.; Adam, G.A.; Hosny, A.; Khodakarami, F.; Waldron, L.; Wang, B.; McIntosh, C.; Goldenberg, A.; Kundaje, A.; Greene, C.S.; et al. Transparency and reproducibility in artificial intelligence. *Nature* **2020**, *586*, E14–E16. [CrossRef]
37. Zhuang, D.; Zhang, X.; Song, S.; Hooker, S. Randomness in neural network training: Characterizing the impact of tooling. *Proc. Mach. Learn. Syst.* **2022**, *4*, 316–336.
38. Qian, S.; Pham, V.H.; Lutellier, T.; Hu, Z.; Kim, J.; Tan, L.; Yu, Y.; Chen, J.; Shah, S. Are my deep learning systems fair? An empirical study of fixed-seed training. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 30211–30227.
39. Morovati, M.M.; Nikanjam, A.; Khomh, F.; Ming, Z. Bugs in Machine Learning-based Systems: A Faultload Benchmark. *arXiv* **2022**, arXiv:2206.12311.
40. Jia, L.; Zhong, H.; Wang, X.; Huang, L.; Lu, X. The symptoms, causes, and repairs of bugs inside a deep learning library. *J. Syst. Softw.* **2021**, *177*, 110935. [CrossRef]
41. Amershi, S.; Begel, A.; Bird, C.; DeLine, R.; Gall, H.; Kamar, E.; Nagappan, N.; Nushi, B.; Zimmermann, T. Software engineering for machine learning: A case study. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 25–31 May 2019; pp. 291–300.
42. Giray, G. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *J. Syst. Softw.* **2021**, *180*, 111031. [CrossRef]
43. Keras Tuner. 2019. Available online: https://github.com/keras-team/keras-tuner (accessed on 1 August 2022).