Studying the Impact of TensorFlow and PyTorch Bindings on Machine Learning Software Quality

HAO LI*, University of Alberta, Canada
GOPI KRISHNAN RAJBAHADUR, Centre for Software Excellence, Huawei Canada, Canada
COR-PAUL BEZEMER*, University of Alberta, Canada

Bindings for machine learning frameworks (such as TensorFlow and PyTorch) allow developers to integrate a framework's functionality using a programming language different from the framework's default language (usually Python). In this paper, we study the impact of using TensorFlow and PyTorch bindings in C#, Rust, Python and JavaScript on the software quality in terms of correctness (training and test accuracy) and time cost (training and inference time) when training and performing inference on five widely used deep learning models. Our experiments show that a model can be trained in one binding and used for inference in another binding for the same framework without losing accuracy. Our study is the first to show that using a non-default binding can help improve machine learning software quality from the time cost perspective compared to the default Python binding while still achieving the same level of correctness.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software performance**; **Correctness**; • **Computing methodologies** → *Neural networks.*

Additional Key Words and Phrases: Software engineering for machine learning, Software quality, Deep learning, Binding, TensorFlow, PyTorch

ACM Reference Format:
Hao Li, Gopi Krishnan Rajbahadur, and Cor-Paul Bezemer. 2024. Studying the Impact of TensorFlow and PyTorch Bindings on Machine Learning Software Quality. *ACM Trans. Softw. Eng. Methodol.* , (2024), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rapidly improving capabilities of Deep Learning (DL) and Machine Learning (ML) frameworks have been the main drivers that allow new intelligent software applications, such as self-driving cars [27, 61] and robotic surgeons [18, 77, 82]. These intelligent software systems all contain components that integrate one or more complex DL and/or ML algorithms. Fortunately, over the past decade, the need for coding these ML and DL algorithms from scratch has been largely eliminated by the availability of several mature ML frameworks and tools such as TensorFlow [1] and PyTorch [63]. These frameworks provide developers with a high-level interface to integrate ML functionality into their projects. Using such ML frameworks has several advantages including

*Hao Li and Cor-Paul Bezemer are with the Analytics of Software, GAmes And Repository Data (ASGAARD) Lab, University of Alberta, Canada.

Authors' addresses: Hao Li, li.hao@ualberta.ca, University of Alberta, Edmonton, AB, Canada, T6G 2R3; Gopi Krishnan Rajbahadur, Centre for Software Excellence, Huawei Canada, Kingston, ON, Canada, K7L 1H3, gopi.krishnan.rajbahadur1@huawei.com; Cor-Paul Bezemer, bezemer@ualberta.ca, University of Alberta, Edmonton, AB, Canada, T6G 2R3.

1049-331X/2024/0-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

readily usable state-of-the-art algorithms, accelerated computing, and interactive visualization tools for data [60].

ML frameworks are typically accessed using Python, which is now the most popular programming language for ML applications [4, 60, 68]. Gonzalez et al. [23] show that more than 56% of the ML projects on GitHub are written in Python. However, many software projects do not use Python as their primary language[1] and the developers of these projects might be unfamiliar with Python. Since learning a new language is a non-trivial task even for experienced developers [76], these developers have to use a workaround to use the Python ML frameworks in their preferred programming language.

To help non-Python developers with the integration of an ML framework, 25% of the popular ML frameworks offer one or more *bindings* for other programming languages [47]. These bindings expose the functionality of the framework in the binding's language. For example, TensorFlow provides a JavaScript binding[2] that allows developers to integrate ML techniques directly in JavaScript. Because a binding adds an additional layer around the ML framework, it is important to investigate how the quality of the ML software created using these ML frameworks is impacted. For instance, different bindings may take different amounts of time to build a model.[3] In addition, bugs in the bindings can introduce inconsistencies for trained models. For example, TensorFlow's C# binding had different results than the Python binding when loading an already trained model due to incorrectly handling 'tf.keras.activations' functions.[4] However, no one has systematically investigated the impact of using bindings for ML frameworks on the ML software quality; typically, studies focus on the software quality of the ML frameworks themselves [9, 52, 78], or on the impact of the computing device on which the model executes [26].

To illustrate the potential impact and importance of our study, consider the following real-world scenario. Anna's team uses JavaScript as the primary programming language. Since the team lacks ML or Python expertise, they collaborate with the company's ML team to integrate DL techniques into their projects. They are now considering using an ML framework's JavaScript binding for their project. However, they are concerned about how their developed ML software's quality is impacted by the binding; in particular, they are concerned about the correctness and time cost. There are three possible scenarios for integration of the binding that our study can assist with choosing the best option:

- **Integration Scenario 1:** The ML team develops and trains the DL models and ships the pre-trained models to Anna. In this scenario, Anna needs to use the JavaScript binding to load the pre-trained models and perform model inference in her project.
- **Integration Scenario 2:** The ML team assists Anna in training DL models in the project's native language which is JavaScript, allowing Anna to alter and maintain the code more efficiently. After training the DL models, Anna needs to deploy the trained models to the production environment in JavaScript as well.
- **Integration Scenario 3:** Since computational resources for the project are very limited, Anna is also open to a third scenario, in which the ML team assists her in selecting the most efficient combination of training and inference bindings in any language. In this scenario, Anna is willing to hire an expert in the chosen language(s) to help with the integration of the binding(s) as long as the reduction in computational resources is large enough.

[1] https://githut.info
[2] https://github.com/tensorflow/tfjs
[3] As can be seen in this GitHub issue for TensorFlow: https://github.com/tensorflow/tensorflow/issues/55476
[4] https://github.com/SciSharp/TensorFlow.NET/issues/991 and https://github.com/SciSharp/TensorFlow.NET/pull/1001

Therefore, in this paper we study the impact of bindings on two important ML software quality aspects:

- **Correctness:** We evaluate if models trained using different bindings for a given ML framework have the same accuracy. We study (1) training accuracy, which captures the model's classification performance on the train set during the training process, and (2) test accuracy, which captures the classification performance of the final trained model on the test set. In addition, we measure whether the test accuracy is the same after loading a pre-trained model in a binding that was not used to train the model (the *cross-binding* test accuracy).
- **Time cost:** We evaluate if models trained using different bindings for an ML framework take similar time for training and making inferences. Bindings that produce models with a high time cost are expensive (in terms of computational resources), which limits their applicability.

We conducted model training and model inference experiments using bindings for TensorFlow and PyTorch in C#, Rust, Python, and JavaScript. In the model training experiments, we trained LeNet-1, LeNet-5, VGG-16, LSTM, GRU, and BERT models on the GPU in every binding (excluding BERT which is only trained on the Python bindings) using the same data and as far as possible, the same framework configuration. In the model inference experiments, we loaded pre-trained models and performed inference using every binding on the CPU and GPU. We do so to address the following research questions (RQs), with RQ1 and RQ2 focusing on correctness, and RQ3 and RQ4 focusing on time cost:

RQ1. How do the studied bindings impact the training accuracy and test accuracy of the studied DL models?
During the training process, bindings for the same ML framework can have different training accuracies for the same model as well as varying test accuracy values (2% difference) in the final trained models.

RQ2. How do the studied bindings impact the cross-binding test accuracy of pre-trained models?
The cross-binding test accuracy of the pre-trained models was not impacted by the bindings.

RQ3. How do the studied bindings impact the training time of the studied DL models?
Non-default bindings can be faster than the default Python bindings for ML frameworks. For instance, PyTorch's Python binding has the slowest training time for the studied models; PyTorch's C# binding is more than two times faster than the Python binding in training the LeNet-5 model.

RQ4. How do the studied bindings impact the inference time of pre-trained models?
Bindings can have very different inference times for the same pre-trained model, and the inference time of certain bindings on CPU can be faster than that of other bindings on GPU. For example, TensorFlow's Rust binding can perform inference faster for an LSTM model on CPU than the JavaScript binding on GPU (73.9 vs. 177.7 seconds).

The main contributions of our paper are as follows:

(1) We are the first to study the impact of using different bindings for ML frameworks on the ML software quality in terms of correctness and time cost.
(2) We found that using a non-default binding can help improve ML software quality (from the time cost perspective) compared to the default Python binding of the studied frameworks in certain tasks, while still achieving the same level of correctness.
(3) We provide a replication package [48], which consists of the implementation of the studied ML models in the studied bindings, scripts for running the experiments, and Jupyter Notebooks for analyzing the experiment results.

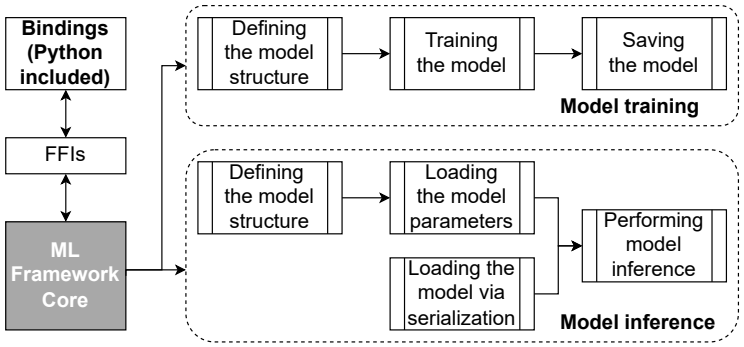

Fig. 1. Bindings use the functionality of ML frameworks via foreign function interfaces (FFIs) to train models and perform model inference.

The remainder of this paper is outlined as follows. Sections 2 provides background information. Section 3 describes the design of our study. Sections 4 and 5 present the results. Section 6 discusses the implications of our findings. Section 7 gives an overview of related work. Section 8 outlines threats to the validity of our study and Section 9 concludes the paper.

2 BACKGROUND

2.1 ML Frameworks

Machine learning frameworks are software libraries that provide ML techniques to developers for the development and deployment of ML systems. Most popular ML frameworks are supported by large companies such as Google and Facebook [4]. As shown in Figure 1, an ML framework provides interfaces to define the structure of a model, train the defined model using a selected optimizer, and save the trained model for later use. In addition, developers can deploy the trained models to the production environment by loading a saved (or *pre-trained*) model and performing inference. ML frameworks can load a pre-trained model using (1) the *model parameters* (e.g., weights and hyperparameters) or (2) *serialization*. If only the model parameters are saved, developers first have to define the model structure before they can load the stored parameters into the defined model. When loading a serialized model, the ML framework can recreate the model from the saved file automatically since it contains both the structure and the weights of the pre-trained model.

Modern ML frameworks, such as TensorFlow and PyTorch, have been built upon a foundation that leverages parallel processing devices like GPUs. GPUs have proven to be highly efficient for tasks that demand parallel computation, especially in the realm of ML. Their architecture is inherently designed to handle multiple tasks simultaneously, allowing for massive parallelism. However, one significant characteristic of GPU computations that needs emphasis is their asynchronous nature. When a task is dispatched to a GPU, it does not always execute immediately. Instead, it often gets scheduled in a queue.[5] Consequently, a CPU might continue with its tasks believing that a GPU job is complete when, in fact, it has not even started. This asynchronous behaviour allows GPUs to optimize task execution but also necessitates careful synchronization when precise timing or task ordering is crucial.

[5]https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

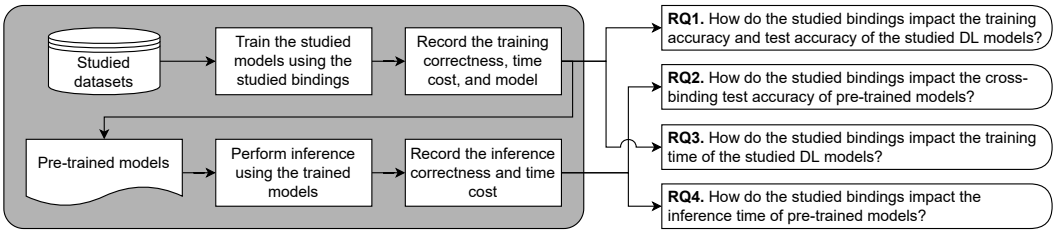

Fig. 2. Overview of the study design.

2.2 Bindings for the ML frameworks

Python is the most popular programming language for ML applications [4, 60], but developers in other languages also have the need for using ML algorithms. Developers might choose an existing ML framework in their preferred language or they have to create a new one from scratch (which requires a large amount of work and is error-prone). Another alternative is to use a *binding* in their preferred language, which provides interfaces to the functionality of an existing ML framework in the language of the binding [47].

As shown in Figure 1, bindings access the functionality of the ML framework through foreign function interfaces (FFIs) without recoding the library. FFIs bridge the gap between programming languages, allowing developers to reuse code from other languages. For example, TensorFlow's Rust binding[6] uses the FFI provided by the Rust language[7] to access TensorFlow functionality. Since the GPU support is provided by the underlying C/C++ computational core of ML frameworks, bindings typically leverage FFIs to access these functionalities. For example, the Python bindings for TensorFlow and PyTorch make use of SWIG[8] (Simplified Wrapper and Interface Generator) and Pybind11[9] to generate FFIs for its Python binding to tap into the C++ backend which includes the ability to access the GPU. However, the efficiency in leveraging GPU resources may vary among different bindings.

3 STUDY DESIGN

In this section, we first describe our experimental environment and the studied datasets, models, ML frameworks, and bindings. Then, we discuss how we evaluate the correctness and time cost in the model training and model inference experiments. Finally, we introduce the experimental setup of our study. Figure 2 gives an overview of our study design.

3.1 Environment setting

We set up our experimental environment on a dedicated laboratory server provided by ISAIC[10], where we can control the execution of other running tasks. The server runs Ubuntu Linux *20.04* with Linux kernel *5.11.0*. We used the CUDA *11.1.74* and cuDNN *8.1.0* GPU-related libraries. The hardware specifications of the server are as follows:

- GPU: 2x NVIDIA TU102 [TITAN RTX] (24 GB)
- CPU: 3.30 GHz Intel(R) Core(TM) i9-9820X
- RAM: 100 GB

[6] https://github.com/tensorflow/rust/tree/master/tensorflow-sys
[7] https://doc.rust-lang.org/rust-by-example/std_misc/ffi.html
[8] https://www.swig.org/
[9] https://github.com/pybind/pybind11
[10] https://isaic.ca/

Table 1. Our studied datasets and models. (Each model is paired with a dataset for the experiments)

Dataset	#Samples*		Model	
	Train	Test	Name	#Parameters
MNIST	60,000	10,000	LeNet-1	4,326
			LeNet-5	61,706
CIFAR-10	50,000	10,000	VGG-16	33,650,890
IMDb	25,000	25,000	LSTM	4,665,537
			GRU	4,250,817
SQuAD	87,599	10,570	BERT (base)	108,893,186

* The split of the training and test set is provided by the dataset.

3.2 Studied datasets and models

Table 1 presents the datasets and models used in this study, specifically pairing each model with the dataset used in the experiments. The datasets we studied are MNIST [45], CIFAR-10 [43], IMDb review [55], and SQuAD [67]. These datasets are widely used as benchmarks in ML research [26, 33, 42, 49, 50, 60, 84, 88]. The models we studied are LeNet [44], VGG [79], LSTM [30], GRU [11], and BERT (the base model) [14] as all of them are typically paired with these datasets in various research domains [2, 12, 21, 26, 31, 33, 75, 83, 84, 90, 91].

MNIST and CIFAR-10 are datasets for image classification tasks. MNIST contains 70,000 grayscale images of handwritten digits, serving as a benchmark for evaluating classification models like LeNet-1 and LeNet-5. We used the CIFAR-10 dataset, which contains 60,000 colour images of 10 different objects, to train the VGG-16 model. The primary metric for these classification tasks is accuracy, reflecting the proportion of correctly identified images out of the total dataset.

The IMDb review dataset is utilized for sentiment analysis (text classification). The dataset contains 25,000 positive and 25,000 negative text reviews of movies. We used it to train the LSTM and GRU models to analyze the sequential nature of text data. Both LSTM and GRU models utilize a recurrent neural network (RNN) structure for handling sequential data, and we integrated a word embedding [13] on the IMDb dataset in our experiments. The performance is measured by accuracy which indicates the model's ability to correctly classify reviews.

SQuAD is a dataset for the extractive question-answering task. SQuAD contains around 100,000 question-answer pairs, where the questions are posed by crowdworkers on a set of Wikipedia articles and the answer to every question is a text span from the corresponding reading passage. We used SQuAD to train the BERT-base model, leveraging the model's capability in language understanding. The task is to identify the exact text span (i.e., start and end positions) within the given passage that answers a question. The evaluation metric for SQuAD is the exact match score [67], which calculates the percentage of questions for which the model's answer exactly matches the annotated answer.

3.3 Studied ML frameworks

We study the latest stable versions (at the time of starting our study) of TensorFlow [1] (*2.5.0*) and PyTorch[11] (*1.9.0*), since they are two of the most popular ML frameworks. TensorFlow and PyTorch

[11] https://github.com/pytorch/pytorch/releases/tag/v0.1.1

Table 2. Studied bindings for TensorFlow and PyTorch in software package ecosystems.

Framework	Name	Ecosystem	Language	Version	# Stars†
TensorFlow	tensorflow	PyPI	Python	2.5.0	177,149
	TensorFlow.NET	NuGet	C#	0.60.4	2,906
	tensorflow	Cargo	Rust	0.17.0	4,627
	@tensorflow/tfjs-node	npm	JavaScript*	3.9.0	17,635
PyTorch	pytorch	PyPI	Python	1.9.0	70,021
	TorchSharp	NuGet	C#	0.93.9	946
	tch	Cargo	Rust	0.5.0	3,178
	@arition/torch-js	npm	JavaScript*	0.12.3	252

* We wrote TypeScript code when using the JavaScript bindings.

† The number of stars on GitHub recorded as of August 24, 2023.

have recently grown in popularity as Caffe2 was merged into PyTorch in 2018[12] and Keras became "the high-level API of TensorFlow 2" [41].

3.4 Studied Bindings

The studied TensorFlow and PyTorch bindings are shown in Table 2. These bindings are all based on the same version of the studied ML frameworks (i.e., TensorFlow *2.5.0* and PyTorch *1.9.0*). Notably, TensorFlow and PyTorch both utilize the Python bindings by default. The reason behind selecting bindings in these four software package ecosystems is twofold: (1) Generally, PyPI (Python), npm (JavaScript), and NuGet (C#) are the three most popular software package ecosystems for cross-ecosystem ML bindings [47] and (2) specifically, the Cargo ecosystem (Rust) is popular (according to the number of stars on GitHub) for both TensorFlow[13] and PyTorch.[14] As shown in Table 2, the number of GitHub stars serves as a proxy for the popularity of a project in the software engineering domain [5, 19, 28, 86, 87], with TensorFlow's JavaScript binding being particularly notable. Although the number of stars for C# and JavaScript bindings for PyTorch may appear low, we included these to ensure a fair comparison with TensorFlow bindings in respective ecosystems.

3.5 Correctness evaluation

Training correctness. During the training process, the correctness is measured in each epoch using the training accuracy which is calculated by $Acc_{train} = N_{correct}/N_{train}$, where $N_{correct}$ is the number of correct predictions and N_{train} is the number of data samples in the training set. For the final trained models, we use the test accuracy $Acc_{test} = N_{correct}/N_{test}$ as the evaluation metric for comparison, which is the accuracy on the test set.

Inference correctness. When we finish training a model, we use the test accuracy Acc_{test} of this pre-trained model as a reference. Then, we perform inference with a studied binding for the pre-trained model on the test set to obtain the cross-binding test accuracy $Acc_{cross\_test} = N_{correct}/N_{test}$ using that binding. The difference between Acc_{test} and Acc_{cross_test} is that the inference correctness is measured in the studied binding. For BERT on SQuAD, we use the exact match score [67] instead of accuracy as the metric to evaluate the correctness.

[12] https://caffe2.ai/

[13] https://github.com/tensorflow/rust

[14] https://github.com/LaurentMazare/tch-rs

Table 3. Supported features of studied bindings for TensorFlow (TF) and PyTorch (PT).

		Training	Supported interfaces			Loading models	
			CNNs	RNNs	BERT	Parameters	Serialization
TF	Python	✓	✓	✓	✓	✓	✓
	C#	✓	✓	✗†	✗	✓	✗
	Rust	✗*	✓	✓	✗	✗	✓
	JavaScript	✓	✓	✓	✗	✗	✓
PT	Python	✓	✓	✓	✓	✓	✓
	C#	✓	✓	✓	✗	✓	✗
	Rust	✓	✓	✓	✗	✓	✓
	JavaScript	✗	✗	✗	✗	✗	✓

* Unlike other bindings, TensorFlow's Rust binding does not support the API (Keras-like) of TensorFlow 2.

† TensorFlow's C# binding has only recently introduced support for RNNs based on TensorFlow 2.10, however, our study uses the C# binding for TensorFlow 2.5.0 for consistency across all bindings.

3.6 Time cost evaluation

Training time cost. The training time cost measures the time spent training a model in seconds. Developers commonly train DL models on GPU rather than CPU since the training can be time-consuming and GPU can considerably shorten the training time [7, 46]. Hence, all model training experiments of bindings for ML frameworks are conducted on GPU and we measure the training time cost on GPU only.

Inference time cost. The inference time cost measures the time spent for performing inference with a pre-trained model on the test set in seconds. Since developers can deploy pre-trained models to a production environment which supports the CPU or GPU, the inference time cost of a binding is measured on both CPU and GPU.

3.7 Experimental setup

In this section, we detail our experimental setup with a running example of how we computed the correctness and time cost of LeNet-1 when trained and inferenced using the studied bindings for the studied ML frameworks.

Step 1 – Train the studied models using the studied bindings: We conduct model training experiments for each supported model-dataset pair (as shown in Table 1). For a given model-dataset pair, each binding that supports the model's interface and training features (as shown in Table 3) trains the model from scratch on that dataset. For example, LeNet-1 and MNIST form one model-dataset pair and each supported binding trains LeNet-1 on MNIST independently. We repeat this process for each model-dataset pair in each binding that supports the model. For consistency, we ensure the following across all bindings for a given model-dataset pair:

- **Model structure.** We use interfaces that provide the same functionality in bindings to build up each layer of the studied models. However, not all bindings support model training, as indicated in Table 3. As a result, we do not conduct training experiments with TensorFlow's Rust binding, PyTorch's JavaScript binding, and RNNs in TensorFlow's C# binding.
- **Training set and test set.** We use the provided split of the training set and test set from studied datasets. Before conducting experiments, we perform comprehensive data preprocessing, ensuring that all bindings can work with the same processed data across all experiments.

Procedure 1 Measuring Training Time Cost in PyTorch Bindings

```
1:  model, optimizer ← initModelAndOptimizer()        ▷ Model and optimizer initialization
2:  train_set ← loadDataset()                         ▷ Load pre-processed training set
3:  start ← getCurrentTime()                          ▷ Start the timer
4:  for epoch ← 1 to epochs do
5:      while not isEndOfDataset(trainSet) do
6:          inputs, labels ← getNextBatch(train_set)      ▷ Batch data loading*1
7:          outputs ← model(inputs)                       ▷ Start forward propagation*2a
8:          loss ← calculateLoss(outputs, labels)         ▷ Loss calculation*2b
9:          loss.backward()                               ▷ Start backward propagation*3a
10:         optimizer.step()                              ▷ Parameter update*3b
11:     end while
12: end for
13: training_time_cost ← getCurrentTime() − start     ▷ Compute elapsed time
14: return training_time_cost
```

*1−3: Subactivities in the training process – forward propagation includes loss calculation and backward propagation includes parameter update.

- **Hyperparameters.** We use the same hyperparameters (e.g., the number of epochs and batch size) and optimizers from prior research [26]. However, TensorFlow's C# binding does not support setting the momentum and weight decay hyperparameters for a stochastic gradient descent (SGD) optimizer. Hence, we only set the learning rate for the SGD optimizer without enabling momentum and weight decaying when training the LeNet-1, LeNet-5, and VGG-16 models to maintain consistency across all bindings. In addition, to mitigate the risk of default hyperparameters influencing our results, we explicitly defined all configurable parameters and kept them the same across bindings.
- **Random seed.** We fix the value of the random seed across bindings when training the same model to control the randomness.

In addition, we repeat the same training process five times for each binding with different random seeds (that are kept consistent across bindings) to reduce the impact of seed selection on the results.

Running example. We train the LeNet-1 model in TensorFlow's Python, C#, and JavaScript bindings. These bindings all set the same random seed at the start of the training process. To build up the same convolution layers of the model, we use the "Conv2D" interface in Python, "Conv2D" in C#, and the "conv2d" interface in JavaScript. In addition, we use SGD with a learning rate of 0.05 for all three bindings to train the LeNet-1 model.

Step 2 – Record the training correctness and save the model: We record the training accuracy in each epoch for all model training experiments. After the training is completed, we compute the trained model's test accuracy and save the model for later use. Considering the impact of randomness, we repeat the training process 5 times in each training experiment and analyze the distribution of the results to draw conclusions.

Running example. During training the LeNet-1 model in PyTorch's C# binding, we calculate the training accuracy in each epoch and store the value. After finishing the training, we save the trained LeNet-1 model.

Step 3 – Perform inference using the trained models and record the inference correctness: For each model inference experiment, each binding loads a pre-trained model via the supported model loading approach(es) (as shown in Table 3) and performs inference on the test set on both CPU and GPU. In addition, bindings for the same ML framework perform inference

Procedure 2 Measuring Inference Time Cost in PyTorch Bindings

```
1: model ← loadSavedModel()                          ▷ Load trained model
2: test_set ← loadDataset()                          ▷ Load pre-processed test set
3: start ← getCurrentTime()                          ▷ Start the timer
4: while not isEndOfDataset(test_set) do
5:     inputs ← getNextBatch(test_set)               ▷ Batch data loading*1
6:     preds ← model(inputs)                         ▷ Inference forward propagation*2
7: end while
8: inference_time_cost ← getCurrentTime() − start    ▷ Compute elapsed time
9: return inference_time_cost
```

*1−2: Subactivities in the inference process.

Procedure 3 Measuring Time Cost of a Training/Inference Subactivity in PyTorch Bindings

```
1: start ← getCurrentTime()                          ▷ Start the timer
2: runSubactivity()                                  ▷ Execute a subactivity of training/inference
3: cuda.synchronize()                                ▷ Wait for the subactivity to finish
4: time_cost ← getCurrentTime() − start              ▷ Compute elapsed time
5: return time_cost
```

for the same pre-trained model. We select the pre-trained models (which are saved in Step 2) from TensorFlow and PyTorch's default Python bindings since the default bindings tend to have the best support and maintenance [47].

Running example. In TensorFlow's Rust binding, we load the pre-trained LeNet-1 model from TensorFlow's default Python binding via serialization to perform model inference on the test set and record the cross-binding test accuracy.

Step 4 – Measure and record the training time cost: Our primary focus is on measuring the time cost of the entire training process on GPU and recording it, as shown in Procedures 1 and 4. Due to the asynchronous nature of GPU computations (as explained in Section 2), we only keep the code directly related to the training process in this step to ensure accurate time measurements, excluding activities like calculating correctness metrics in each epoch (which is included in Steps 1 and 2). We also do not include the time cost of initialization processes, such as model initialization, optimizer initialization, and initial dataset loading.

Procedure 1 within PyTorch showcases its granular control over the training process. It initiates by setting up the model and optimizer, loading the training dataset, and iterating through the epochs for optimizing the model weights. For each epoch, the process starts with loading a batch of the data. Following this, forward propagation is performed to produce outputs which are used for calculating the loss values. Lastly, backward propagation is executed to calculate the gradients which guide the optimizer for updating the model parameters. In contrast, as demonstrated in Procedure 4, TensorFlow offers less granularity since it encapsulates the entire training process (i.e., batch data loading, forward propagation, and backward propagation) within a single function to optimize performance.

As shown in Procedure 3, the granularity control in PyTorch is particularly helpful in measuring time costs for specific subactivities using the "cuda.synchronize()" function to facilitate synchronization between the CPU and GPU. The "cuda.synchronize()" function is only available in the Python and Rust bindings. Procedure 3 starts a timer, runs a subactivity (e.g., forward propagation), waits for the subactivity to finish using "cuda.synchronize()", and then computes the elapsed time.

Procedure 4 Measuring Training/Inference Time Cost in TensorFlow (TF) Bindings

```
model ← initModelAndCompile(optimizer, loss_function)        ▷ Model initialization
train_set, test_set ← loadDataset()                           ▷ Load pre-processed data
start ← getCurrentTime()                                      ▷ Start the timer
model.fit(train_set, epochs)/model.predict(test_set)          ▷ TF's single training/inference function
time_cost ← getCurrentTime() − start                          ▷ Compute elapsed time
return time_cost
```

Running example. We train the LeNet-1 model with PyTorch's Python binding and employ Procedure 1 to record the training time cost. In addition, we rerun the training experiment utilizing Procedure 1 with additional synchronization steps as described in Procedure 3 to capture accurate time costs for individual subactivities.

Step 5 – Measure and record the inference time cost: Similar to Step 4, we measure and record the time cost of the entire inference process on both CPU and GPU following Procedures 2 and 4. For measuring the time costs of inference subactivities (i.e., batch data loading and forward propagation), we rerun the inference experiments employing Procedure 3, but only for PyTorch's Python and Rust bindings on GPU.

Running example. In PyTorch's Python binding, we use Procedure 2 to determine the inference time cost for the pre-trained LeNet-1 model. Furthermore, we rerun the inference experiment with additional steps from Procedure 3 to separately record time costs for batch data loading and forward propagation.

3.8 Supported features in studied bindings

Table 3 outlines the supported features by each studied binding:

- **Training support:** A lack of training support in certain bindings means developers might have to use another programming language. This can be inconvenient and result in additional overhead, especially if developers are unfamiliar with the alternative language.
- **Model interface support:** When certain model types are not supported in a binding, developers might still need to switch to another language to train their models.
- **Model loading approaches:** Loading models via serialization provides flexibility as developers don't need to define the model structure. In contrast, loading models via parameters requires the model's structure to be pre-defined. This can lead to challenges, especially when developers try to use pre-trained models.

For our training experiments in Section 3.7, certain bindings are exempt due to their limitations: TensorFlow's Rust and PyTorch's JavaScript bindings (which don't support training), TensorFlow's C# binding for RNNs, and all bindings for BERT. We acknowledged the recent inclusion of support for RNNs in TensorFlow's C# binding (aligned with TensorFlow v2.10).[15] However, to maintain consistency in our experimental framework, we focused on TensorFlow version 2.5.0 which is the most commonly supported version of TensorFlow by the studied bindings.

For the inference experiments, all bindings are utilized in our work, with the exception of RNNs in TensorFlow's C# and BERT in C# bindings for both ML frameworks. The reason is that the C# bindings can only load models using parameters and lacks support for RNN and BERT interfaces. Unlike PyTorch's JavaScript binding which despite not supporting CNNs, RNNs, and BERT, does offer loading via serialization without the need for defining model structures.

[15] https://github.com/SciSharp/TensorFlow.NET/issues/640

Table 4. Mean/Max DTW distances of training accuracy curves for bindings in training models with the same random seed. (Highlighted numbers indicate negligible DTW distance. Py: Python; JS: JavaScript; Rs: Rust)

Model	TensorFlow (mean/max DTW distance)			PyTorch (mean/max DTW distance)		
	Py-C#	Py-JS	JS-C#	Py-C#	Py-Rs	Rs-C#
LeNet-1	0.005/0.006	**0.000/0.000**	0.005/0.006	**0.000/0.000**	**0.000/0.000**	**0.000/0.000**
LeNet-5	0.003/0.004	**0.000/0.000**	0.003/0.004	**0.000/0.000**	**0.000/0.000**	**0.000/0.000**
VGG-16	0.018/0.019	0.005/0.006	0.018/0.019	0.007/0.010	0.002/0.003	0.008/0.010
LSTM	-	0.008/0.012	-	0.008/0.009	0.009/0.011	0.010/0.011
GRU	-	0.010/0.012	-	0.010/0.011	0.008/0.009	0.009/0.010

4 CORRECTNESS EVALUATION

Motivation. Developers can use a binding for an ML framework in their preferred programming language to train a DL model. We want to observe if the DL models trained using a binding for a given ML framework have the same training accuracy as the DL models trained using the ML framework's default Python binding (RQ1). These results can help developers understand if using a binding will achieve the same model accuracy during training and provide the same model performance for the final trained models.

In addition, it is important to ascertain if performing inference for these trained models using different bindings for a given framework will impact the accuracy. Pre-trained models have been widely used by the ML community [29, 85] and bindings can help developers to run inference with pre-trained models in different programming languages. Importantly, in high-stakes domains such as medical diagnosis and autonomous driving, accuracy is particularly important when decisions are made by ML systems [62]. Even a slight drop in accuracy can trigger erroneous decisions with serious implications. Hence, it is vital that bindings have the capability to achieve the same accuracy for pre-trained models as with the binding they were trained with. In RQ2, we investigate the cross-binding test accuracy of pre-trained models using the bindings for TensorFlow and PyTorch to understand whether the pre-trained models perform as we would expect them to.

Together, the bindings' impact on training correctness and inference correctness will enable us to understand the impact on the correctness of the ML software quality.

RQ1: How do the studied bindings impact the training accuracy and test accuracy of the studied DL models?

Approach. We employ both dynamic time warping (DTW) [72] for analyzing training accuracy curves and the Mann-Whitney U test [56] for comparing the performance metrics of the final trained models. We chose DTW due to its ability to analyze time-series data, which allows us to investigate whether different bindings follow the same trajectory during training. DTW calculates the distance between the training accuracy curves of the bindings (e.g., between TensorFlow's Python and C# binding) for training the same model. DTW is widely used as a distance measurement for time series data since it can manage time distortion by aligning two time series before computing the distance, which is more accurate than the Euclidean distance [15]. We normalize the calculated DTW distances between 0 to 1 to interpret the results. A normalized DTW distance of 0 means that the difference between the two curves is negligible.

In addition, we calculate the test accuracy, F1-score, and AUC-ROC for the final trained models to compare their classification performance. For each metric, we perform the Mann-Whitney U test [56] separately at a significance level of $\alpha = 0.05$ to determine if the values obtained from

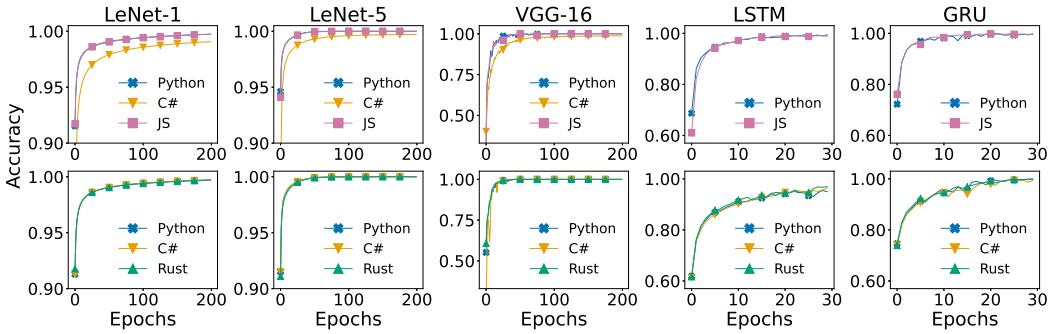

Fig. 3. Mean training accuracy curves of LeNet-1, LeNet-5, VGG-16, LSTM, and GRU on GPU in bindings for TensorFlow (first row) and PyTorch (second row).

different bindings are significantly different. We computed Cliff's delta d [53] effect size to quantify the difference based on the following thresholds [70]:

$$\text{Effect size} = \begin{cases} \textit{negligible}, & \text{if } |d| \leq 0.147 \\ \textit{small}, & \text{if } 0.147 < |d| \leq 0.33 \\ \textit{medium}, & \text{if } 0.33 < |d| \leq 0.474 \\ \textit{large}, & \text{if } 0.474 < |d| \leq 1 \end{cases} \tag{1}$$

Findings. Bindings can have different training accuracy curves when training DL models under the same configuration (i.e., model structure, training data, hyperparameters, and random seed). Table 4 reports the mean and maximum DTW distances for the training curves between bindings. Moreover, Figure 3 presents the mean training accuracy curves of the models (out of the five training processes) that have the best test accuracy after the last epoch. The figure and table show that bindings can have quite different training accuracy curves according to the DTW distance when using the same training configuration. For example, the distances between the curves of TensorFlow's C# binding and the other two bindings are relatively large for LeNet-1, LeNet-5, and VGG-16 models. Another example is that all PyTorch bindings have a relatively large distance between the curves for the RNN models compared to the distances in the CNN models. One reason could be the differential numerical precision across programming languages. For example, Python supports arbitrary-precision arithmetic, while languages like Rust and C# typically operate with fixed precision. These variations in numerical precision might spawn minor differences in mathematical computation outputs. These minor differences might accumulate over numerous iterations during model training, resulting in variations in the final model accuracy. In contrast, bindings can exhibit nearly the same behaviour for training some DL models; the training accuracy curves of the LeNet models differ negligibly between TensorFlow's Python and JavaScript bindings, as well as between PyTorch's bindings.

The trained models produced by certain bindings can perform worse than the models produced by other bindings for the same ML framework. Table 5 shows the test accuracy, F1-score, and AUC-ROC for the trained models produced by bindings can be different. For the trained VGG-16 models, the Mann-Whitney U test reveals significant differences between bindings for both frameworks in these metrics with large effect sizes. This pattern is also observed in the trained GRU models in PyTorch's bindings. Specifically, while the test accuracy and F1-score of the trained LeNet-1 models have statistically significant differences between bindings for TensorFlow, the AUC-ROC values of LeNet models in TensorFlow and PyTorch bindings are close (all rounded

Table 5. The average test accuracy (Acc), F1-score (F1), and AUC-ROC (AUC) for TensorFlow and PyTorch bindings. (Statistically significant differences between bindings are highlighted in bold. Py: Python; JS: JavaScript; Rs: Rust; MD: Max Diff; ES: Effect Size)

		TensorFlow						PyTorch				
		LN1	LN5	VGG	LSTM	GRU		LN1	LN5	VGG	LSTM	GRU
Acc	Py	98.8	98.9	84.8	83.7	85.0	Py	98.8	98.9	86.2	86.5	87.9
	C#	98.6	98.9	83.8			C#	98.8	99.0	86.2	87.3	85.5
	JS	98.8	99.0	85.6	84.2	84.7	Rs	98.8	98.9	85.6	87.4	87.0
	MD	**0.2**	0.1	**1.9**	0.6	0.3	MD	0.0	0.1	**0.6**	0.8	**2.5**
	p	**0.01**	0.40	**0.01**	0.10	0.15	*p*	0.68	0.31	**0.03**	0.10	**0.01**
	ES	**large**	-	**large**	-	-	ES	-	-	**large**	-	**large**
F1	Py	98.8	98.9	84.7	83.5	85.0	Py	98.8	99.0	86.3	86.7	87.9
	C#	98.6	98.9	83.8			C#	98.8	99.0	86.1	87.2	85.1
	JS	98.8	99.0	85.6	83.8	84.7	Rs	98.9	98.9	85.6	87.2	86.9
	MD	**0.2**	0.1	**1.9**	0.3	0.3	MD	0.1	**0.1**	**0.7**	0.5	**2.8**
	p	**0.01**	0.42	**0.01**	0.22	0.15	*p*	0.06	**0.01**	**0.01**	0.10	**0.01**
	ES	**large**	-	**large**	-	-	ES	-	**large**	**large**	-	**large**
AUC	Py	100.0	100.0	98.2	91.7	92.3	Py	100.0	100.0	98.5	94.1	94.3
	C#	100.0	100.0	97.3			C#	100.0	100.0	98.5	94.6	92.9
	JS	100.0	100.0	98.4	92.3	91.9	Rs	100.0	100.0	98.3	94.5	93.8
	MD	0.0	0.0	**1.1**	**0.6**	**0.5**	MD	0.0	0.0	**0.2**	0.5	**1.5**
	p	0.10	0.84	**0.01**	**0.01**	**0.01**	*p*	0.55	0.42	**0.01**	0.10	**0.01**
	ES	-	-	**large**	**large**	**large**	ES	-	-	**large**	-	**large**

up to 100 in Table 5). Furthermore, we observed some models produced by non-Python bindings have higher values of the metrics than the models produced by the default Python bindings, e.g., the VGG-16 model produced by TensorFlow's JavaScript binding.

Summary of RQ1

TensorFlow and PyTorch bindings can have different training accuracy curves for training the same DL models even when using the same configuration. In addition, the test accuracy of the final trained models can be slightly different. Hence, developers should not assume that all bindings offer the same level of correctness and should verify the model's correctness when utilizing a binding for training.

RQ2: How do the studied bindings impact the cross-binding test accuracy of pre-trained models?

Approach. We conducted inference experiments with all bindings using pre-trained models produced by the default Python bindings for TensorFlow and PyTorch (see Figure 4). We loaded the pre-trained models using the supported loading approach(es) and recorded the cross-binding test accuracy on both CPU and GPU for each binding. If the cross-binding test accuracy of a pre-trained model in a binding shows a 0% difference compared to the test accuracy when the model was initially trained, we considered the test accuracy "reproduced" by that binding. Any non-zero difference resulted in a "failed" mark. Since some bindings only support one way of loading models (as shown

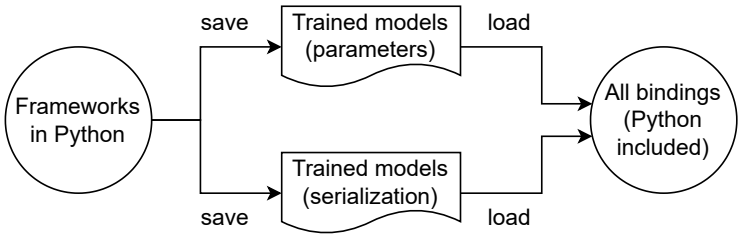

Fig. 4. All bindings load the trained models that are saved by the default Python bindings for ML frameworks.

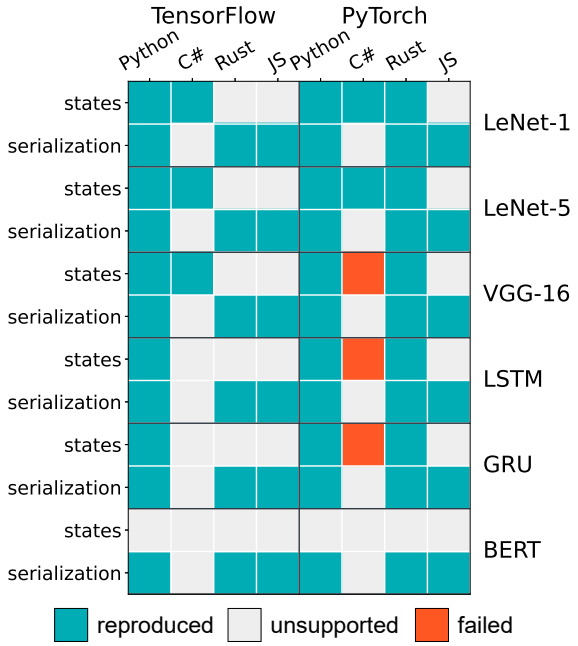

Fig. 5. Results of reproducing the test accuracy of pre-trained models in TensorFlow and PyTorch bindings on the CPU and GPU (the results are identical). Note: the failed cases in the PyTorch's C# binding were fixed in a newer version of the binding.

in Table 3), we marked the result as "unsupported" if the loading approach is not supported by a binding.

Findings. The test accuracy of pre-trained models can be reproduced across bindings in different languages for the same ML framework. Figure 5 shows that only PyTorch's C# binding failed to reproduce the test accuracy in the saved VGG-16, LSTM, and GRU models. We noticed that the differences in the test accuracy in these three models are all within 1% and the root cause of the reproduction failure is a bug that results in "eval() and train() methods not being properly propagated to all submodules".[16] This bug prevents setting the model to evaluation mode, hence, the dropout layers of these three models are not disabled which leads to different cross-binding test accuracy. This bug is fixed in version *0.96.0* which does not support PyTorch *1.9.0*

[16]See https://github.com/dotnet/TorchSharp/pull/501 and https://github.com/dotnet/TorchSharp/issues/500

but targets version *1.10.0*. In other words, the saved models can be reproduced in the newer version of PyTorch's C# binding. For consistency, we still use the *0.93.9* version of this binding for the other experiments.

Bindings can reproduce the test accuracy of pre-trained models via different loading approaches and on different types of processing units (i.e., CPU and GPU). As shown in Figure 5, PyTorch's Python and Rust bindings and TensorFlow's Python binding support both loading via parameters and serialization, and both loading approaches can reproduce the test accuracy of the pre-trained models. In addition, we noticed that bindings can reproduce the test accuracy of pre-trained models on both CPU and GPU.

Summary of RQ2

TensorFlow and PyTorch bindings can perform inference using pre-trained models and reproduce the same test accuracy as when the models were originally trained. This correctness property holds true whether model inference is performed on CPU or GPU. As a result, developers can leverage the capabilities of pre-trained models while still being able to use the model in their preferred language.

5 TIME COST EVALUATION

Motivation. In RQ1 and RQ2, we studied the impact of bindings for ML frameworks on correctness, however, the impact of bindings on time cost remains unknown. Given the time-consuming nature of model training and model inference for ML frameworks, it is important to investigate how a binding may impact the time cost. Studies show that runtime efficiency and energy consumption can vary across programming languages [59, 64, 66]. Consequently, these differences may have an impact on the time cost of training and inference when using different bindings.

Thus, in RQ3, we study the time cost of training DL models with bindings in order to offer developers more information about the overhead or advantage in terms of time cost when training with a binding. In RQ4, we study the inference time of pre-trained models in bindings. The time of utilizing bindings in model inference can be a crucial consideration for developers since model inference typically takes place (as a part of the product) in the production environment, which may have limited resources. The findings can help developers decide whether or not to utilize a binding for model inference in their project.

RQ3: How do the studied bindings impact the training time of the studied DL models?

Approach. To study the difference in training time across bindings, we performed the Mann-Whitney U test [56] using the Bonferroni correction [74] to adjust the significance level for multiple comparisons. Specifically, for an initial significance level of $\alpha = 0.05$, we adjusted the significance level to $\frac{\alpha}{n}$ (where n is the number of comparisons made) to determine whether the distributions of the training times of the default Python bindings and the non-Python bindings, which trained the same model for the same framework, are significantly different. For example, the LeNet-1 model in TensorFlow bindings, we performed Bonferroni-corrected Mann-Whitney U test between the Python and C# bindings and Python and JavaScript bindings with an adjusted significance level of $\frac{\alpha}{2} = 0.025$. We also computed Cliff's delta d [53] effect size to quantify the difference based on Equation 1 in Section 4.

Findings. Training times can differ greatly across bindings for the same ML framework. Figure 6 shows the training time distributions on GPU for the studied models across the studied bindings. The Bonferroni-corrected Mann-Whitney U test shows that the training time distributions

Fig. 6. Training time distributions when training models in TensorFlow and PyTorch bindings on the GPU.

of the same model are all significantly different between the default Python bindings and the other bindings for the same framework and the effect sizes are all large. In addition, the difference in training time of bindings for the same ML framework can be very large when training certain models. For example, the median training time of TensorFlow's JavaScript binding for the VGG-16 model is 15 times larger than its Python binding (32,783 vs. 1,991 seconds).

PyTorch's default Python binding has the slowest training time for the studied models. Figure 6 shows that PyTorch's Python binding is more than two times slower than the other two bindings for training LeNet models. However, we note that the training time difference between PyTorch's Python binding and other bindings for the VGG-16, LSTM, and GRU models is relatively small (less than 15%). In contrast, TensorFlow's default Python binding has the fastest training time in the studied models.

Batch data loading time affects the training cost of PyTorch's Python binding. As shown in Table 6, PyTorch's Python binding has a long batch data loading time, which is notably slower (between 4 to 14 times) than the Rust binding for all studied models. Specifically, For LeNet models, the Python binding's batch data loading times account for roughly 30% of the training cost, whereas the Rust binding's batch data loading for the same models consumes less than 10% of the training cost. Furthermore, the Python binding consistently underperforms the Rust binding during both forward and backward propagation phases in the studied models.

The observed variations in batch data loading times between bindings suggest that the native speed of a programming language [59, 64, 66] is an important factor that influences the performance of a binding. However, there could be other factors involved in the implementation of

Table 6. Time costs (in seconds) of the subactivities in the training process using PyTorch's Python and Rust bindings on GPU.

		Load batch data	Forward	Backward	Total
LeNet-1	Python	148.9	76.2	240.2	465.7
	Rust	23.5	69.7	239.2	332.5
LeNet-5	Python	167.4	114.8	293.2	576.4
	Rust	24.1	94.3	278.8	397.4
VGG-16	Python	89.2	7094.0	1557.8	8741.2
	Rust	31.9	6470.4	1469.5	7971.8
LSTM	Python	8.2	95.4	188.8	292.2
	Rust	0.6	83.5	165.3	249.4
GRU	Python	8.5	84.0	151.0	242.8
	Rust	0.6	72.5	130.3	203.5

bindings. For example, these factors could include overheads arising from differences in data structure implementations and initialization routines. Additionally, the overhead of the marshalling mechanism [6, 16, 89] implemented to convert data between the binding's programming language and the ML framework could impact efficiency. Finally, the way the binding interacts with the ML framework's lower-level APIs, such as those for memory management and tensor operations, could also play a crucial role in performance differences.

Summary of RQ3

Training times for training the same DL models differ significantly between the default Python bindings and the non-Python bindings for the same ML framework. Surprisingly, non-Python bindings for PyTorch are even faster in training the studied models than the default Python binding. Hence, choosing the right binding can help developers to lower the training time cost for certain models.

RQ4: How do the studied bindings impact the inference time of pre-trained models?

Approach. We followed the same process as shown in Figure 4 and investigated the inference time of each model on both CPU and GPU. We performed the Bonferroni-corrected Mann-Whitney U test on the recorded inference time distributions between the default Python bindings and the non-Python bindings, grouped by the same framework, model, and processing unit (CPU or GPU). We also computed Cliff's Delta effect size as described in RQ3.

Findings. The inference time of the same pre-trained model differs greatly between the default Python bindings and the other bindings for the same ML framework. Figure 7 shows the distributions of the inference time of the pre-trained models in the studied bindings. The results of the Bonferroni-corrected Mann-Whitney U test and Cliff's Delta d show that the Python and non-Python bindings for the same ML framework have significantly different inference times for the same model on the same processing unit (i.e., CPU and GPU) and the effect size is large, except for the TensorFlow bindings for LSTM on CPU and for BERT on GPU where the Python binding has similar inference time costs as the Rust binding. We observed that the default Python bindings for TensorFlow and PyTorch do not always offer the best inference time for all studied

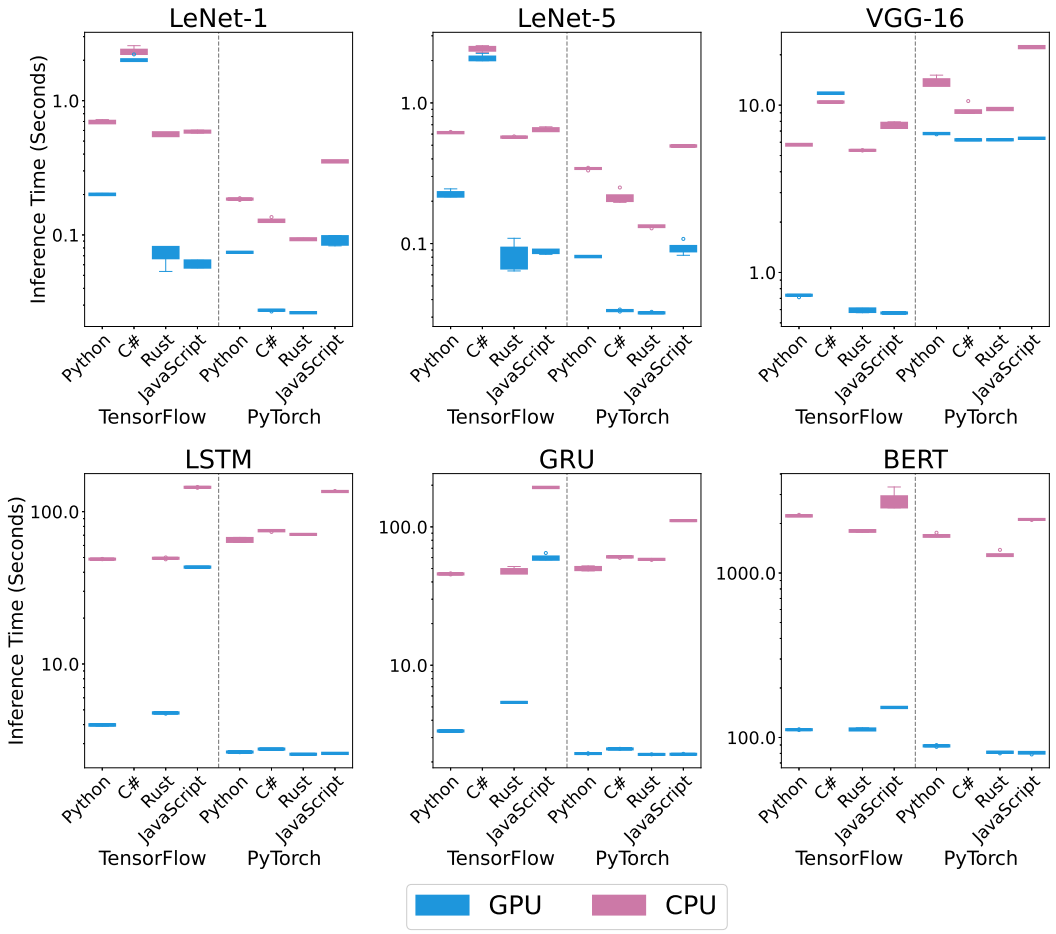

Fig. 7. Inference time distributions for pre-trained models in TensorFlow (TF) and PyTorch (PT) bindings on the CPU and GPU.

pre-trained models, with Rust bindings often outperforming them. On the other hand, TensorFlow's C# binding has the worst performance for the studied models on both CPU and GPU, and PyTorch's JavaScript binding has the worst performance on CPU. Moreover, the performance gap in model inference time can be very large, for example, TensorFlow's Python binding is 17 times as fast as the JavaScript binding for the GRU model on the GPU (3.35 vs. 58.32 seconds).

Inference time differences in PyTorch arise from both batch data loading and forward propagation speed. Table 7 shows that the majority of the inference cost is allocated towards forward propagation and the Rust binding outperforms the Python binding in this regard. As we observed the same pattern in RQ3, the Rust binding also demonstrates faster batch data loading times compared to the Python binding across all studied models. Although both bindings leverage PyTorch's computational core, which is written in C/C++ and predominantly runs computations on GPUs, the variations in time costs can be attributed to overheads introduced by the bindings themselves.

Table 7. Time costs (in seconds) of the subactivities in the inference process using PyTorch's Python and Rust bindings on GPU.

		Load batch data	Forward	Total
LeNet-1	Python	0.06	0.02	0.08
	Rust	0.01	0.02	0.03
LeNet-5	Python	0.06	0.03	0.08
	Rust	0.01	0.02	0.03
VGG-16	Python	0.12	6.74	6.86
	Rust	0.03	6.26	6.29
LSTM	Python	0.15	2.61	2.76
	Rust	0.01	2.57	2.58
GRU	Python	0.15	2.27	2.41
	Rust	0.03	2.25	2.28
BERT	Python	0.13	88.86	88.99
	Rust	0.12	81.82	81.94

Certain bindings on the CPU may have a faster inference time than other bindings on the GPU for the same pre-trained model. Generally, inference time for pre-trained models on GPU outperforms CPU in bindings for both studied frameworks (as shown in Figure 7). However, we found that for the same framework, one binding that runs inference on CPU can outperform another binding that runs on GPU for the same pre-trained model. For example, the Rust binding for TensorFlow is faster on CPU than the C# binding on GPU for LeNet and VGG-16 models, as well as faster on CPU than the JavaScript binding on GPU for GRU model. Furthermore, we noticed that TensorFlow's C# binding in model inference on CPU is similar to or even faster than on GPU. According to the maintainer of the C# binding, the reason could be that "there is I/O cost underlying"[17] model inference on GPU.

Certain bindings lack support for certain features which leads to a slower inference time. We noticed that TensorFlow's JavaScript binding cannot load a GRU model with "reset_after=True"[18], either by loading parameters or through serialization. However, "reset_after=True" is the default setting in the framework (and other bindings) to enable the "fast cuDNN implementation", which speeds up the inference of the GRU model[19] This unsupported feature can be one of the reasons behind the large increase of GRU inference time in TensorFlow's JavaScript binding (256.5 seconds) compared to the inference time of the default Python binding (3.6 seconds).

Summary of RQ4

TensorFlow and PyTorch bindings have various inference times for the same pre-trained models on CPU and GPU. Remarkably, the inference time of certain models in bindings on the CPU can be faster than other bindings for the same framework on GPU. Therefore, developers can experiment and choose the fastest binding for their usage scenario.

[17] https://github.com/SciSharp/TensorFlow.NET/issues/876
[18] https://github.com/tensorflow/tfjs/issues/4621
[19] https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU

6 IMPLICATIONS

6.1 Implications for developers

Developers are not limited to writing their projects in Python when using an ML framework. Although Python dominates the development in ML [4, 60], developers can also use bindings in other programming languages. Our results in Section 4 shows that non-default bindings for TensorFlow and PyTorch can have the same inference accuracy of a pre-trained model as the default Python binding and sometimes even faster performance. We recommend developers use the binding in their preferred programming language for either model training or inference if supported by the binding. Hence, developers can save time and effort when adopting ML techniques in their projects without having to settle for non-mature ML frameworks that might be available in the language that their current software is programmed in. For instance, in Integration Scenario 1 of Section 1, Anna can use the JavaScript binding to perform inference with pre-trained models provided by the ML team.

Developers can use a binding for an ML framework which has a shorter training time for a certain model and perform inference on the trained model in another binding that has a shorter inference time based on task and requirements. Bindings for an ML framework have various training times and inference times for ML models (Section 5). Hence, developers can choose different bindings which are faster for a certain model in training and inference respectively since the accuracy of pre-trained models can be reproduced across bindings for the same framework (Section 4). We suggest that developers refer to an existing benchmark like ours or conduct experiments themselves based on our replication package [48]. For example, when using TensorFlow for LeNet models as described in Integration Scenario 3 of Section 1, Anna can train the models using the default Python binding for TensorFlow and then run inference for the trained model in the Rust binding with the assistance of a hired expert to save time and computational resources, as this factor is critical in their project requirements.

Developers should perform a sanity check before using a model that was trained by a binding other than the default Python binding. Bindings corresponding to different languages can have different training accuracy curves while training the same model, and the final trained model can behave differently (as discussed in Section 4). Since the Python bindings are the default binding for most ML frameworks, these Python bindings have a larger user base and better support than other bindings. We suggest that developers perform a sanity check on the trained model if they are using a binding other than the default Python binding before deploying the models to the production environment.

In resource-limited scenarios (e.g., CPU only), developers may prefer or need to use a non-default binding for model inference. Traditionally, model inference is done using a GPU due to the superior inference time of GPUs [7, 46]. However, GPUs are expensive and not available in all scenarios. We found that the bindings for ML frameworks can be fast for running inference on CPU for some pre-trained models (Section 5). Developers can use such bindings if the production environment does not contain a GPU or the computational resource is limited. For example, in Integration Scenario 3 of Section 1, if Anna is using PyTorch for LeNet models and there is no GPU available in the production environment, she can use PyTorch's Rust binding on CPU with expert assistance. The inference time of LeNet models in the Rust binding on CPU is faster than the default Python binding both on CPU and GPU. This is particularly beneficial for constrained environments like the Internet-of-Things (IoT) devices (e.g., unmanned aerial vehicles) where resource availability is often limited [3, 20, 40, 71].

6.2 Implications for binding owners

Binding owners should include performance benchmarks for their binding. We found that bindings can have very different training and inference times for ML models (Section 5), yet this information is not well documented. To address this, we suggest that binding owners introduce performance benchmarks of training and running inference for some frequently used ML models (e.g., VGG models) and record the results in their documentation. This way, developers be aware of the trade-off between choosing a familiar language and the potential impact on time cost for various DL models. For example, the performance benchmarks can help Anna in Integration Scenario 2 of Section 1 to make informed decisions when choosing a familiar language for training while considering the potential impact on time cost.

6.3 Implications for researchers

Researchers should investigate the impact of ML framework bindings on large-scale models and datasets. Our findings provide a starting point, but further research is needed to fully understand how binding choices influence performance in large-scale models. While full-parameter fine-tuning can be computationally expensive, parameter-efficient techniques like Low-Rank Adaptation (LoRA) [32] offer a cost-effective alternative. However, LoRA's experimental status in HuggingFace[20] and its lack of binding support highlight a direction for further research. We suggest future research adopt our methodology (see our replication package [48]), starting with representative data subsets and smaller model variants (e.g., the 7 billion parameter variant of Llama 2 [81]). This approach could provide valuable early insights into potential performance variations before committing to full-scale experiments.

Researchers should investigate methods to enhance the interoperability and compatibility of pre-trained models across different bindings for ML frameworks. Our findings demonstrate that pre-trained models can be used across different bindings for the same ML framework with the same level of accuracy (as shown in Section 4). However, some models may not be supported or may have a slower inference time when utilizing certain bindings (as discussed in Section 5). While developers and binding owners focus on the implementation of bindings, we suggest researchers explore ways to contribute at a higher level: by devising algorithms, methodologies, or protocols to increase the interoperability and compatibility of pre-trained models across different bindings, benefiting a diverse developer base.

Researchers should study the patterns and origins of bugs in bindings for ML frameworks. We found that bugs in bindings for ML frameworks have an impact on the model inference correctness (Section 4). While the immediate resolution of bugs in bindings is an engineering concern, a deeper analysis of these issues can provide invaluable insights into software design and testing paradigms for bindings. Although researchers have previously studied bugs in ML frameworks [9, 38, 39], there has been no research specifically on bugs in the bindings for ML frameworks or other libraries. We encourage researchers to systematically analyze the bugs in bindings and provide guidelines for maintainers to avoid introducing such bugs.

7 RELATED WORK

7.1 Impact of ML frameworks on ML software correctness

Researchers have studied the correctness of ML frameworks. However, no one has studied how bindings for those frameworks impact the correctness of the ML software that is created with them. The study by Guo et al. [26] is the closest related to our work. However, even though they included several bindings in their study, their work differs from ours as they focus on the impact on ML

[20] https://huggingface.co/docs/diffusers/en/training/lora

software quality of using different ML frameworks and executing ML models on different computing devices (such as PC and various types of mobile devices). In contrast, we run our experiments on the same device but we study the impact of various bindings on ML software quality. Hence, we can reason about the impact of using a binding, while in Guo et al.'s study, the different devices make this impossible.

Several others have focused on comparing the accuracy of the same model across ML frameworks. Chirodea et al. [10] compared a CNN model that was built with TensorFlow and PyTorch and found that these two frameworks have similar training curves but the final trained model has a lower accuracy in PyTorch. Gevorkyan et al. [22] gave an overview of five ML frameworks and compared the accuracy of training a neural network for the MNIST dataset. They reported that the final trained model has a lower accuracy in TensorFlow than in other frameworks. Moreover, Elshawi et al. [17] conducted training experiments for six ML frameworks by using the default configuration and reported that certain frameworks have better performance than the other frameworks on the same model (e.g., Chainer on the LSTM model).

7.2 Impact of ML frameworks on ML software time cost

Many studies have compared the time cost across ML frameworks. In a comparison of the training and inference time for a CNN architecture using PyTorch and TensorFlow, Chirodea et al. [10] found that PyTorch is faster in both model training and inference than TensorFlow. However, Gevorkyan et al. [22] showed that PyTorch has the worst training time for neural networks among five studied ML frameworks. In our work, we compared the training and inference time across bindings for the same ML frameworks.

Several studies have focused on the time cost of ML frameworks on different hardware devices. Buber and Diri [7] compared the running time of DL models on CPU and GPU and found that GPU is faster. Jain et al. [37] focused on the performance of training DNN models on CPU with TensorFlow and PyTorch. They show that multi-processing provides better training performance when using a single-node CPU. For mobile and embedded devices, Luo et al. [54] introduced a benchmark suite to evaluate the inference time cost based on six different neural networks.

7.3 Impact of ML frameworks on ML software reproducibility

Reproducibility has become a challenge in ML research [25, 57, 80]. Liu et al. [51] surveyed 141 published ML papers and conducted experiments for four ML models. The results showed that most studies do not provide a replication package and the models are highly sensitive to the size of test data. In addition, Isdahl and Gundersen [35] introduced a framework to evaluate the support of reproducing experiments in ML platforms and found that the platforms which have the most users have a relatively lower score in reproducibility. In this paper, we studied the reproducibility of pre-trained models across different bindings for the same ML framework.

To improve the reproducibility of ML models, many researchers have conducted studies to understand and resolve non-deterministic factors in ML software. Pham et al. [65] studied nondeterminism-introducing-factors in ML frameworks (e.g., weight initialization and parallel processes) and found that these factors can cause a 10% accuracy difference in ML models. To improve the reproducibility of ML models, Chen et al. [8] suggested using patching to minimize nondeterminism in hardware and proposed a record-and-reply approach to eliminate randomness in software. In addition, they provided guidelines for producing a reproducible ML model. Nagarajan et al. [58] studied deterministic implementation for deep reinforcement learning and proposed a deterministic implementation of deep Q-learning by identifying and controlling five common sources of nondeterminism.

7.4 Empirical Studies of ML Frameworks

Many empirical studies of ML frameworks exist that study software quality aspects such as software bugs [9, 38, 39], technical debt [52, 73], and programming issues [34, 36, 92]. However, no prior work has investigated the impact of bindings for ML frameworks on the ML software quality.

Many studies have focused on the bugs of ML frameworks. Jia et al. [38, 39] investigated TensorFlow's GitHub repository and identified six symptoms and eleven root causes of bugs in TensorFlow. In addition, they found that most bugs are related to interfaces and algorithms. Chen et al. [9] studied bugs from four ML frameworks and investigated the testing techniques in these frameworks. They showed that the most common root cause of the bugs is the incorrect implementation of algorithms, and the current testing techniques have a low percentage of test coverage.

ML software has ML-specific technical debts such as unstable data dependence, hidden feedback loop, and model configuration debts [73]. This technical debt can hurt the maintainability of ML systems and introduce extra costs. Liu et al. [52] analyzed self-admitted technical debt in 7 DL frameworks and concluded that technical debt is common in DL frameworks, although application developers are often unaware of its presence.

Researchers have also aimed to understand the ML frameworks from a developer perspective to study the programming issues when using an ML framework. They typically researched the questions and answers (Q&As) of developers about ML frameworks on Stack Overflow (SO). Zhang et al. [92] investigated Q&As which are related to TensorFlow, PyTorch and Deeplearning4j on SO and reported that model migration is one of the most frequently asked questions. Humbatova et al. [34] studied Q&As of these three ML frameworks on SO as well and included interviews with developers and researchers to build a taxonomy of faults in ML systems. Islam et al. [36] mined Q&As about ten ML frameworks on SO and reported that developers need both static and dynamic analysis tools to help fix errors.

7.5 FFIs and Bindings in Software Engineering

FFIs and language bindings are instrumental in software engineering, serving as bridges that enable different programming languages to collaborate seamlessly. These bridges often enable developers to develop applications in their language of choice while simultaneously using mature libraries that are developed in another language. The existing body of work predominantly proposes approaches to design and improve such bindings and FFIs within one specific language. For instance, Yallop et al. [89] conducted experiments to create bindings for using the *ctypes* library in OCaml. Their study differentiated the performance of dynamic and static bindings, revealing that static bindings could be between 10 to 65 times faster than their dynamic counterparts. This finding aligns with our investigation into the time costs associated with diverse ML software bindings.

Researchers also proposed several approaches to FFIs. For instance, Bruni et al. [6] introduced an FFI approach called NativeBoost. This approach requires minimal virtual machine modifications and generates native code directly at the language level. They compared the time cost of different FFIs and the results show that NativeBoost is competitive. Ekblad et al. [16] presented an FFI tailored for web-targeting Haskell dialects, emphasizing simplicity and automated marshalling. The authors compare their FFI with the vanilla FFI, which is based on C calling conventions, and show that their FFI has some advantages in terms of simplicity and expressiveness, safety, without introducing excessive performance (i.e., time cost) overhead.

In addition, Ravitch et al. [69] automated the generation of library bindings using static analysis, aiming to simplify the often laborious manual creation process. Their method not only refined the automated binding generation but also unveiled type bugs in manually created bindings,

highlighting potential threats to software correctness. Meanwhile, Grimmer [24] explored high-performance language interoperability in multi-language runtimes. Their approach leveraged just-in-time (JIT) compilers to optimize across language borders, enhancing the efficiency of cross-language operations.

To the best of our knowledge, our study is the first to systematically investigate the impact of using different language bindings on ML software quality. While Ravitch et al. [69] touched upon type correctness in bindings, the unique challenges posed by the inherently non-deterministic nature of ML software remain under-explored. Our work stands out as we specifically evaluate the impact of bindings on the correctness of ML software for model training and inference across different languages. In addition, The computationally intensive nature of ML software introduces unique challenges when assessing time costs, especially when relying on GPUs. While time cost is a widely used metric in the domain of FFIs and bindings, existing works do not explore its significance within the context of ML frameworks. Our research actively fills this void, presenting a comprehensive analysis of time costs associated with different bindings in ML software on CPUs and GPUs.

8 THREATS TO VALIDITY

8.1 Construct validity

We use the accuracy metric to assess the correctness of TensorFlow and PyTorch bindings on model training and inference since it is a widely used metric among researchers and developers [10, 17, 22, 26, 54]. However, other metrics may also be used to assess correctness and use of other metrics could potentially change our results. For evaluating the time cost of bindings on model training, we ran training experiments on the GPU since training DL models on CPU is time-consuming and developers usually train DL models on GPU. The results might be different from those obtained by measuring the time cost on CPU.

8.2 Internal validity

When implementing the studied models in TensorFlow and PyTorch bindings, we used the same/similar interfaces to ensure that the structures of these models are consistent across bindings. However, bindings might have different implementations for these interfaces (or have hidden bugs) that result in different structures in the built models. We saved the built models in bindings (via parameters or serialization) and loaded them back into the default Python bindings for TensorFlow and PyTorch to examine whether the structures were the same. The verification results confirm that the produced models in bindings have the same structures.

TensorFlow's JavaScript binding does not support training and inference for GRU with "reset_after=True". Hence, we set "reset_after=False" in the training experiment of TensorFlow's JavaScript binding for GRU and performed inference with a GRU model that was trained with "reset_after=False" in the default Python binding. This setup differs from other bindings, although it has no effect on the model's structure. We compared the results from the JavaScript binding to the results in the Python binding using "reset_after=False", and our findings still hold. Future studies should investigate how one can automatically confirm that the configurations of the bindings are exactly the same.

8.3 External validity

We focused on TensorFlow and PyTorch bindings in our work and the results of our study might not apply directly to other ML frameworks. One reason could be that other ML frameworks could have a different implementation and do not provide GPU support. Furthermore, the findings of our

investigation may not be able to generalize to other models and datasets. Future studies should leverage our methodology to analyze bindings for other ML frameworks using different models and datasets.

Our analysis focused on small to medium-sized models that are widely adopted in real-world applications. However, the implications for large-scale models, particularly frontier ML models with billions or trillions of parameters, require further investigation. Future research should build on our work to examine how the observed differences might persist or change at this extreme scale.

9 CONCLUSION

In this paper, we investigate the impact on ML software quality (correctness and time cost) of using bindings for ML frameworks for DL model training and inference. We conducted model training and model inference experiments on three CNN-based models and two RNN-based models in TensorFlow and PyTorch bindings written in four different programming languages. The most important findings of our study are:

- When training models, bindings for ML frameworks can have various training accuracy curves and slightly different test accuracy values for the trained models.
- Bindings have different training times for the same model, and the default Python bindings for ML frameworks may not have the fastest training time.
- Bindings for ML frameworks have the capabilities to reproduce the accuracy of pre-trained models for inference.
- Bindings for ML frameworks have different inference times for the same pre-trained model and certain models in bindings on the CPU can outperform other bindings on the GPU.

Our findings show that developers can utilize a binding to speed up the training time for an ML model. For pre-trained models, developers can perform inference in their favoured programming language without sacrificing accuracy, or they can choose a binding that has better inference time.

DISCLAIMER

Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of Huawei.

ACKNOWLEDGMENTS

The work described in this paper has been supported by the ECE-Huawei Research Initiative (HERI) at the University of Alberta.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, USA, 265–283.

[2] Saheed Salahudeen Abdullahi, Sun Yiming, Shamsuddeen Hassan Muhammad, Abdulrasheed Mustapha, Ahmad Muhammad Aminu, Abdulkadir Abdullahi, Musa Bello, and Saminu Mohammad Aliyu. 2021. Deep Sequence Models for Text Classification Tasks. In *International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. 1–6. https://doi.org/10.1109/ICECCE52056.2021.9514261

[3] Andrea Albanese, Matteo Nardello, and Davide Brunelli. 2022. Low-power deep learning edge computing platform for resource constrained lightweight compact UAVs. *Sustainable Computing: Informatics and Systems* 34 (2022), 100725. https://doi.org/10.1016/j.suscom.2022.100725

[4] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden)

(MSR '18). Association for Computing Machinery, New York, NY, USA, 353–363. https://doi.org/10.1145/3196398.3196445

[5] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344. https://doi.org/10.1109/ICSME.2016.31

[6] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Luc Fabresse. 2013. Language-side foreign function interfaces with nativeboost. In *International Workshop on Smalltalk Technologies*.

[7] Ebubekir Buber and Banu Diri. 2018. Performance Analysis and CPU vs GPU Comparison for Deep Learning. In *2018 6th International Conference on Control Engineering Information Technology (CEIT)*. 1–6. https://doi.org/10.1109/CEIT.2018.8751930

[8] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming (Jack) Jiang. 2022. Towards Training Reproducible Deep Learning Models. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2202–2214. https://doi.org/10.1145/3510003.3510163

[9] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (Sept. 2023), 31 pages. https://doi.org/10.1145/3587155

[10] Mihai Cristian Chirodea, Ovidiu Constantin Novac, Cornelia Mihaela Novac, Nicu Bizon, Mihai Oproescu, and Cornelia Emilia Gordan. 2021. Comparison of Tensorflow and PyTorch in Convolutional Neural Network-based Applications. In *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. 1–6. https://doi.org/10.1109/ECAI52376.2021.9515098

[11] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, Doha, Qatar, 103–111. https://doi.org/10.3115/v1/W14-4012

[12] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast Changeset-Based Bug Localization with BERT. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 946–957. https://doi.org/10.1145/3510003.3510042

[13] Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning* (Helsinki, Finland) *(ICML '08)*. Association for Computing Machinery, New York, NY, USA, 160–167. https://doi.org/10.1145/1390156.1390177

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[15] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1542–1552. https://doi.org/10.14778/1454159.1454226

[16] Anton Ekblad. 2015. Foreign Exchange at Low, Low Rates a Lightweight FFI for Web-Targeting Haskell Dialects. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages* (Koblenz, Germany) *(IFL '15)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/2897336.2897338

[17] Radwa Elshawi, Abdul Wahab, Ahmed Barnawi, and Sherif Sakr. 2021. DLBench: a comprehensive experimental evaluation of deep learning frameworks. *Cluster Computing* 24, 3 (Sept. 2021), 2017–2038. https://doi.org/10.1007/s10586-021-03240-4

[18] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. 2019. A guide to deep learning in healthcare. *Nature medicine* 25, 1 (2019), 24–29.

[19] Hongbo Fang, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. 2022. "This is Damn Slick!": Estimating the Impact of Tweets on Open Source Project Popularity and New Contributors. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. 2116–2129. https://doi.org/10.1145/3510003.3510121

[20] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[21] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. 2020. Linear Mode Connectivity and the Lottery Ticket Hypothesis. In *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119. PMLR,

3259–3269.

[22] Migran N Gevorkyan, Anastasia V Demidova, Tatiana S Demidova, and Anton A Sobolev. 2019. Review and comparative analysis of machine learning libraries for machine learning. *Discrete and Continuous Models and Applied Computational Science* 27, 4 (Dec. 2019), 305–315. https://doi.org/10.22363/2658-4670-2019-27-4-305-315

[23] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. The state of the ML-universe: 10 years of artificial intelligence & machine learning software development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 431–442.

[24] Matthias Grimmer. 2014. High-Performance Language Interoperability in Multi-Language Runtimes. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity* (Portland, Oregon, USA) *(SPLASH '14)*. Association for Computing Machinery, New York, NY, USA, 17–19. https://doi.org/10.1145/2660252.2660256

[25] Odd Erik Gundersen and Sigbjørn Kjensmo. 2018. State of the Art: Reproducibility in Artificial Intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (Apr. 2018).

[26] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 810–822. https://doi.org/10.1109/ASE.2019.00080

[27] Abhishek Gupta, Alagan Anpalagan, Ling Guan, and Ahmed Shaharyar Khwaja. 2021. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array* 10 (2021), 100057. https://doi.org/10.1016/j.array.2021.100057

[28] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. 2019. Characterization and Prediction of Popular Projects on GitHub. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 21–26. https://doi.org/10.1109/COMPSAC.2019.00013

[29] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[30] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[31] Oumaima Hourrane, Nouhaila Idrissi, and El Habib Benlahmar. 2019. An Empirical Study of Deep Neural Networks Models for Sentiment Classification on Movie Reviews. In *1st International Conference on Smart Systems and Data Science (ICSSD)*. 1–6. https://doi.org/10.1109/ICSSD47982.2019.9003171

[32] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

[33] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Lei Ma, Mike Papadakis, and Yves Le Traon. 2022. An Empirical Study on Data Distribution-Aware Test Selection for Deep Learning Enhancement. *ACM Transactions on Software Engineering and Methodology* (Jan. 2022). https://doi.org/10.1145/3511598

[34] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[35] Richard Isdahl and Odd Erik Gundersen. 2019. Out-of-the-Box Reproducibility: A Survey of Machine Learning Platforms. In *15th International Conference on eScience (eScience)*. 86–95. https://doi.org/10.1109/eScience.2019.00017

[36] Md Johirul Islam, Hoan Anh Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. What Do Developers Ask About ML Libraries? A Large-scale Study Using Stack Overflow. *arXiv preprint arXiv:1906.11940* (June 2019). arXiv:1906.11940

[37] Arpan Jain, Ammar Ahmad Awan, Quentin Anthony, Hari Subramoni, and Dhableswar K. DK Panda. 2019. Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters. In *IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. https://doi.org/10.1109/CLUSTER.2019.8891042

[38] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *Database Systems for Advanced Applications*, Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.). Springer International Publishing, Cham, 604–620.

[39] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935. https://doi.org/10.1016/j.jss.2021.110935

[40] Mohammed Jouhari, Abdulla Khalid Al-Ali, Emna Baccour, Amr Mohamed, Aiman Erbad, Mohsen Guizani, and Mounir Hamdi. 2022. Distributed CNN Inference on Resource-Constrained UAVs for Surveillance Systems: Design and Optimization. *IEEE Internet of Things Journal* 9, 2 (2022), 1227–1242. https://doi.org/10.1109/JIOT.2021.3079164

[41] Keras. 2021. *About Keras*. Retrieved March 28, 2022 from https://keras.io/about/

[42] Serhat Kiliçarslan and Mete Celik. 2021. RSigELU: A nonlinear activation function for deep neural networks. *Expert Systems with Applications* 174 (2021), 114805. https://doi.org/10.1016/j.eswa.2021.114805
[43] Alex Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images (Technical Report). (April 2012).
[44] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791
[45] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. *The MNIST database of handwritten digits.* Retrieved March 28, 2022 from http://yann.lecun.com/exdb/mnist/
[46] Feng Li, Yunming Ye, Zhaoyang Tian, and Xiaofeng Zhang. 2019. CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms. *Neural Computing and Applications* 31, 8 (Aug. 2019), 4353–4365. https://doi.org/10.1007/s00521-018-3354-z
[47] Hao Li and Cor-Paul Bezemer. 2022. Studying Popular Open Source Machine Learning Libraries and Their Cross-Ecosystem Bindings. *arXiv preprint arXiv:2201.07201* (Jan. 2022). https://doi.org/10.48550/ARXIV.2201.07201
[48] Hao Li, Gopi Krishnan Rajbahadur, and Cor-Paul Bezemer. 2024. *The replication package of our study on bindings for TensorFlow and PyTorch.* https://github.com/asgaardlab/CmpMLBindings
[49] Xiaoyun Li, Belhal Karimi, and Ping Li. 2022. On Distributed Adaptive Optimization with Gradient Compression. In *International Conference on Learning Representations*.
[50] Enlu Lin, Qiong Chen, and Xiaoming Qi. 2020. Deep reinforcement learning for imbalanced classification. *Applied Intelligence* 50, 8 (Aug. 2020), 2488–2502. https://doi.org/10.1007/s10489-020-01637-z
[51] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the Reproducibility and Replicability of Deep Learning in Software Engineering. *ACM Transactions on Software Engineering and Methodology* 31, 1, Article 15 (Oct. 2021), 46 pages. https://doi.org/10.1145/3477535
[52] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3377815.3381377
[53] Jeffrey D. Long, Du Feng, and Norman Cliff. 2003. Ordinal Analysis of Behavioral Data. In *Handbook of Psychology*, Irving B. Weiner (Ed.). John Wiley & Sons, Inc., Hoboken, NJ, USA, Chapter 25, 635–661. https://doi.org/10.1002/0471264385.wei0225
[54] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. 2020. Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices. *arXiv preprint arXiv:2005.05085* (May 2020).
[55] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (Portland, Oregon) *(HLT '11)*. Association for Computational Linguistics, USA, 142–150.
[56] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18 (1947), 50–60.
[57] Matthew B. A. McDermott, Shirly Wang, Nikki Marinsek, Rajesh Ranganath, Luca Foschini, and Marzyeh Ghassemi. 2021. Reproducibility in machine learning for health research: Still a ways to go. *Science Translational Medicine* 13, 586 (2021), eabb1655. https://doi.org/10.1126/scitranslmed.abb1655
[58] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2019. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. In *AAAI 2019 Workshop on Reproducible AI.* https://doi.org/10.48550/ARXIV.1809.05676
[59] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 778–788. https://doi.org/10.1109/ICSE.2015.90
[60] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Alvaro Lopez Garcia, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52, 1 (June 2019), 77–124. https://doi.org/10.1007/s10462-018-09679-z
[61] Jianjun Ni, Yinan Chen, Yan Chen, Jinxiu Zhu, Deena Ali, and Weidong Cao. 2020. A Survey on Theories and Applications for Self-Driving Cars Based on Deep Learning Methods. *Applied Sciences* 10, 8 (2020). https://doi.org/10.3390/app10082749
[62] Anne-Marie Nussberger, Lan Luo, L Elisa Celis, and Molly J Crockett. 2022. Public attitudes value interpretability but prioritize accuracy in Artificial Intelligence. *Nature communications* 13, 1 (2022), 5821.
[63] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*. Curran Associates, Inc., 8024–8035.

[64] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[65] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. 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 Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 771–783. https://doi.org/10.1145/3324884.3416545

[66] L. Prechelt. 2000. An empirical comparison of seven programming languages. *Computer* 33, 10 (2000), 23–29. https://doi.org/10.1109/2.876288

[67] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2383–2392. https://doi.org/10.18653/v1/D16-1264

[68] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information* 11, 4 (2020), 193.

[69] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. 2009. Automatic Generation of Library Bindings Using Static Analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 352–362. https://doi.org/10.1145/1542476.1542516

[70] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*. Citeseer, 1–51.

[71] Arish S., Sharad Sinha, and Smitha K.G. 2019. Optimization of Convolutional Neural Networks on Resource Constrained Devices. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 19–24. https://doi.org/10.1109/ISVLSI.2019.00013

[72] Stan Salvador and Philip Chan. 2007. FastDTW: Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis* 11, 5 (2007), 561–580.

[73] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc.

[74] Juliet Popper Shaffer. 1995. Multiple hypothesis testing. *Annual review of psychology* 46, 1 (1995), 561–584.

[75] Kedi Shen, Yun Zhang, Lingfeng Bao, Zhiyuan Wan, Zhuorong Li, and Minghui Wu. 2023. Patchmatch: A Tool for Locating Patches of Open Source Project Vulnerabilities. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 175–179. https://doi.org/10.1109/ICSE-Companion58688.2023.00049

[76] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?. In *IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 691–701.

[77] Alexey A. Shvets, Alexander Rakhlin, Alexandr A. Kalinin, and Vladimir I. Iglovikov. 2018. Automatic Instrument Segmentation in Robot-Assisted Surgery using Deep Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 624–628. https://doi.org/10.1109/ICMLA.2018.00100

[78] Julien Siebert, Lisa Joeckel, Jens Heidrich, Adam Trendowicz, Koji Nakamichi, Kyoko Ohashi, Isao Namba, Rieko Yamamoto, and Mikio Aoyama. 2022. Construction of a quality model for machine learning systems. *Software Quality Journal* 30, 2 (June 2022), 307–335. https://doi.org/10.1007/s11219-021-09557-y

[79] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations (ICLR 2015)*.

[80] Rachael Tatman, J. Vanderplas, and Sohier Dane. 2018. A Practical Taxonomy of Reproducibility for Machine Learning Research. In *Reproducibility in Machine Learning Workshop at ICML 2018*.

[81] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian,

Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[82] Beatrice van Amsterdam, Matthew J. Clarkson, and Danail Stoyanov. 2021. Gesture Recognition in Robotic Surgery: A Review. *IEEE Transactions on Biomedical Engineering* 68, 6 (2021), 2021–2035. https://doi.org/10.1109/TBME.2021.3054828

[83] Benyou Wang, Lifeng Shang, Christina Lioma, Xin Jiang, Hao Yang, Qun Liu, and Jakob Grue Simonsen. 2021. On Position Embeddings in BERT. In *International Conference on Learning Representations*.

[84] Kang Wang, Yong Dou, Tao Sun, Peng Qiao, and Dong Wen. 2022. An automatic learning rate decay strategy for stochastic gradient descent optimization methods in neural networks. *International Journal of Intelligent Systems* (2022). https://doi.org/10.1002/int.22883

[85] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[86] Thomas Wolter, Ann Barcomb, Dirk Riehle, and Nikolay Harutyunyan. 2023. Open Source License Inconsistencies on GitHub. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 110 (July 2023), 23 pages. https://doi.org/10.1145/3571852

[87] Xiaoya Xia, Shengyu Zhao, Xinran Zhang, Zehua Lou, Wei Wang, and Fenglin Bi. 2023. Understanding the Archived Projects on GitHub. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 13–24. https://doi.org/10.1109/SANER56733.2023.00012

[88] Dongpo Xu, Shengdong Zhang, Huisheng Zhang, and Danilo P. Mandic. 2021. Convergence of the RMSProp deep learning method with penalty for nonconvex optimization. *Neural Networks* 139 (2021), 17–23. https://doi.org/10.1016/j.neunet.2021.02.011

[89] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Science of Computer Programming* 164 (2018), 82–97. https://doi.org/10.1016/j.scico.2017.04.002 Special issue of selected papers from FLOPS 2016.

[90] Zhanglu Yan, Jun Zhou, and Weng-Fai Wong. 2021. Near Lossless Transfer Learning for Spiking Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 12 (May 2021), 10577–10584.

[91] Chengran Yang, Bowen Xu, Jiakun Liu, and David Lo. 2023. TECHSUMBOT: A Stack Overflow Answer Summarization Tool for Technical Query. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 132–135. https://doi.org/10.1109/ICSE-Companion58688.2023.00040

[92] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020