When Good and Reproducible Results are a Giant with Feet of Clay:  
The Importance of Software Quality in NLP

Sara Papi, Marco Gaido, Andrea Pilzer, Matteo Negri  
Fondazione Bruno Kessler  
University of Trento  
NVIDIA  
{spapi,mgaido,negri}@fbk.eu,apilzer@nvidia.com

Abstract

Despite its crucial role in research experiments, code correctness is often presumed solely based on the perceived quality of results. This assumption, however, comes with the risk of erroneous outcomes and, in turn, potentially misleading findings. To mitigate this risk, we posit that the current focus on reproducibility should go hand in hand with the emphasis on software quality. We support our arguments with a case study in which we identify and fix three bugs in widely used implementations of the state-of-the-art Conformer architecture. Through experiments on speech recognition and translation in various languages, we demonstrate that the presence of bugs does not prevent the achievement of good and reproducible results, which however can lead to incorrect conclusions that potentially misguide future research. As countermeasures, we release pancolinNN, a library dedicated to testing neural models, and propose a Code-quality Checklist, with the goal of promoting coding best practices and improving software quality within the NLP community.

1 Introduction

In the field of natural language processing (NLP), as well as in broader contexts, the validity and soundness of research findings are typically upheld by “establishing consistency of results versus existing implementations, standard benchmarks, or sanity checks via statistically significant experimental results” (Rozier and Rozier, 2014). Embracing these recommendations as the exclusive criteria for validating scientific credibility, the research community has recently devoted significant attention to reproducibility alone does not “guarantee the quality, correctness, or validity of the published results” since the code employed to produce them may not accurately execute its intended purpose. This entails inherent risks, as flawed code that produces good and easily reproducible results can propagate as the foundation for further research, ultimately leading to further unreliable and potentially misleading findings (McCullough et al., 2008).

Expanding on these observations, this paper is a call to action, underpinned by empirical evidence, to bolster the dependability of NLP findings by complementing current initiatives toward reproducibility and experimental soundness with equal emphasis on software quality. To this end, we adopt as a reference framework the principles of software quality assurance (SQA – Buckley and Poston 1984; Tripathy and Naik 2011), which have so far been overlooked by our community. Building on this foundation, we contribute as follows:

1. We examine the extent to which research works consider the attributes studied in the SQA field ($\S 2$), highlighting that code correctness has been neglected by the NLP community thus far ($\S 3$);
2. Through a case study on open-source implementations of the widespread Conformer architecture (Gulati et al., 2020), we show that:

- At least one impactful bug is present in all the analyzed implementations (§4.1);
- Such bugs do not prevent from achieving good and reproducible results that outperform other architectures in speech recognition and translation across different language pairs (§4.3);
- Undetected bugs can lead to erroneous conclusions when evaluating new techniques (§4.4).

3. We release a bug-free implementation of Conformer, along with all the pre-trained models;

4. We promote code correctness and software quality by releasing pangolinN, a library featuring easily-usable unit tests to enforce the proper behavior of neural models (§5.1), and proposing the integration into current conferences checklists of a Code-quality section, which would focus on coding best practices (§5.2).

2 SQA and Research

Software Quality Assurance (SQA) attributes have been studied for many years (McCall et al., 1977; Deutsch and Willis, 1988; Glass, 1992). Delineated in the ISO 9126 standard (ISO/IEC, 2001), they were later extended and superseded by ISO 25010 (ISO/IEC, 2010), having production code as the main target. However, as they are desirable for any codebase, here we analyze how each attribute has an effect on research code and work.

Portability and usability refer, respectively, to the possibility of executing the same experiments in diverse hardware or software environments, and the effort required to use the software (i.e., how easy it is to run the code). As such, they pertain to the reproducibility of a paper, which, according to ACM, holds when “an independent group can obtain the same result using the author’s own artifacts”, a definition that is aligned with those given in NLP (Ulmer et al., 2022) and other fields (Schloss, 2018). Regarding these aspects, ample literature already discussed the need to go beyond code openness in research (Chen et al., 2019; Trisovic et al., 2022), highlighting the role of proper documentation and validation in different environments. However, as many research groups lack access to a wide range of hardware options, we argue that research works can hardly target portability due to the significant economic and human resources it requires. On the contrary, proper documentation of the code is a reasonable demand and, in addition to increasing reproducibility, it facilitates code reuse and adoption for other works.

Code reusability also pertains to the maintainability attribute, which denotes the effort required for implementing targeted modifications. Alongside comprehensive documentation, software maintainability hinges on code structure, i.e. the organization of the software into building blocks (Perry and Wolf, 1992; Garlan and Shaw, 1993). While there is currently no incentive to develop reusable code (Barba, 2019), the research community would greatly benefit in the long term from a commitment to this objective, which would reduce the time spent in replicating prior work and accelerate the implementation of new techniques upon existing code.

The expeditiousness of testing new methods also depends on the efficiency and reliability of the codebase. Efficiency refers to the amount of resources a software uses, e.g. the number of GPU hours or VRAM GBs needed for training. Increasing efficiency constitutes a research direction on its own and can hardly be considered a prerequisite for orthogonal investigations. Reliability, instead, is the capability of the software to seamlessly operate in all conditions and for a long time: software causing frequent crashes (i.e. terminations due to errors) or whose efficiency is not constant over time is not reliable. Although both properties would contribute to reducing the environmental footprint of NLP research by avoiding computing-resource wastes or unexpected failures (Strubell et al., 2019; Shterionov and Vanmassenhove, 2023), a commitment in this direction is arguably an excessive demand for research works not expressly dedicated to it.

Last but not least, functionality or functional correctness (hereinafter: correctness) pertains to the “extent to which a program satisfies its specifications” (McCall et al., 1977). In research, this holds when the code exactly performs the operations described in a paper, thereby establishing the validity of the reported findings. Achieving correctness requires the creation and execution of tests, as they are the sole mechanism that guarantees the correct behavior of software. For example, when designing a causal model (i.e. a model that can-
not look at future elements in the input sequence), researchers should test that the model predictions always remain unaffected by future elements. If a bug breaks the causality property, any observed gains may not stem from the proposed solutions but from undue access to forbidden information. It is worth emphasizing that the validity of these tests expires after any code alteration, regardless of its apparent relevance. Therefore, tests should be executed after each modification to ensure correctness and, in turn, the trustworthiness of the findings.

In summary, we have observed that, in the context of research software, (i) portability and usability support reproducibility, (ii) maintainability promotes reusability, (iii) efficiency and reliability reduce environmental costs, and (iv) correctness plays a crucial role in ensuring trustworthy findings and research soundness. However, despite its importance, we show in the next section that the research community has largely neglected correctness, focusing primarily on reproducibility.

3 Research Code Quality Evaluation

To assess the level of consideration given to the above SQA attributes within the NLP research community, we examined their inclusion in the review forms of top-tier conferences and journals in the field, namely: *ACL (i.e., AACL-IJCNLP, ACL, EACL, EMNLP, NAACL), 5 ACL Rolling Review (ARR), ICASSP, ICML, ICLR, Interspeech, NeurIPS, and TACL. We specifically focused on reproducibility (as a proxy of portability and usability) and correctness. Table 1 shows the results.

Most of the venues (5 out of 8) include an explicit score for reproducibility and NeurIPS mentions it among the factors contributing to the overall recommendation score. Reproducibility is commonly evaluated through dedicated checklists6 that mainly focus on the detailed descriptions of the hyperparameters and the software/hardware environment (while disregarding whether different hardware/software is supported, i.e. portability, which seems reasonable as seen in §2). Accordingly, these checklists are not strictly related to SQA, although they do include recommendations for proper code documentation, which is related to the software usability and maintainability.

Table 1: Reproducibility and correctness in the review forms of major NLP conferences/journals.

<table>
<thead>
<tr>
<th>Venue</th>
<th>Reproducibility</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>*ACL</td>
<td>✅</td>
<td>✗</td>
</tr>
<tr>
<td>ARR</td>
<td>✅</td>
<td>✗</td>
</tr>
<tr>
<td>ICASSP</td>
<td>✗</td>
<td>✅</td>
</tr>
<tr>
<td>ICML</td>
<td>✗</td>
<td>✅</td>
</tr>
<tr>
<td>ICLR</td>
<td>✅</td>
<td>✗</td>
</tr>
<tr>
<td>Interspeech</td>
<td>✅</td>
<td>✗</td>
</tr>
<tr>
<td>NeurIPS</td>
<td>✗</td>
<td>✅</td>
</tr>
<tr>
<td>TACL</td>
<td>✅</td>
<td>✗</td>
</tr>
</tbody>
</table>

Correctness is instead mentioned in fewer forms (3 out of 8). When present, its definition varies and is not explicitly related to the code: at ICLR and ICASSP, the scope of the term is not clearly defined, while in TACL it is included in the broader concept of soundness of the experiments/results. In this result-oriented definition, soundness pertains to assessing the significance of results (are the reported improvements robust to statistical fluctuations?) with respect to either the state of the art (are the results competitive with those reported in recent literature?) or strong baselines. Soundness is also assessed at NeurIPS and ICML but, again, code correctness is never explicitly mentioned. Notably, the Interspeech form contains a “Technical Correctness” score, which however refers to the reproducibility of the paper (“are enough details provided to be able to reproduce the experiments?”). In general, when considered, software is explicitly evaluated not only in terms of accessibility (is the code released open-source?) and potential usefulness (will the research community benefit from the use of the software?), but also for its correctness, thus being again more related to its usability and maintainability.

We can conclude that, unlike reproducibility-related SQA attributes, correctness is largely neglected in favor of a result-based evaluation of soundness. From the researchers’ perspective, this entails the risk of basing future work on unreliable software that yields high and easily reproducible results but lacks guarantees of its correctness. This risk, in turn, can lead to misleading findings (McCullough et al., 2008). In the next section, we present a concrete instance of this problem with a case study analyzing open-source implementations of the widespread Conformer architecture.

---

5Since EACL 2024, *ACL conferences adopt ARR only.
4 The Case Study

In our case study, we examine the Conformer (Gulati et al., 2020) architecture – the state-of-the-art solution for speech processing tasks (Guo et al., 2021; Ma et al., 2021; Srivastava et al., 2022; Li and Doddipatla, 2023) such as automatic speech recognition (ASR) and speech-to-text translation (ST) – whose rapid and wide adoption is evidenced by more than 2,000 citations since 2020.7

In the following, we first analyze the Conformer implementation of six widely-used open-source codebases, showing that they all contain at least one bug (§4.1). Then, through extensive experiments on the two tasks and on eight language pairs, we demonstrate that the presence of bugs can be hidden by good – but incorrect – results (§4.3), consequently leading to erroneous conclusions (§4.4).

An introduction of the ASR and ST tasks object of our study, along with an overview of the Conformer architecture is provided in Appendix A.

4.1 Analysis of the Codebases

We analyze the behavior of the open-source implementations of the Conformer by systematically varying a parameter that should not affect the results: the inference batch size (IBS). With high IBSs, many samples are collected in the same batch, allowing for their parallel processing on GPU to reduce the overall computational cost. When samples of different lengths are collected in the same batch – a frequent situation in speech tasks, where the input length largely varies – the input sequences are brought to the same dimension by filling them with padding. Since with correct implementations the results are independent of the presence of padding (and, therefore, of the IBS), research papers usually include only the training batch size (which, instead, is an important hyperparameter for the stability of the training). However, as we demonstrate in this section, the bugs present in the Conformer implementations undermine the above assumption.

We studied six widely-used repositories, namely: Fairseq-ST (Wang et al., 2020), ESPnet-ST (Inaguma et al., 2020), NeMo (Kuchaiev et al., 2019), SpeechBrain (Ravanelli et al., 2021), an open source codebase named “Conformer”,8 and TorchAudio (Yang et al., 2021). We discovered that all these implementations return different results with different IBSs, showing that the presence of padding incorrectly alters the results.9 Upon inspection of the codes, we isolated three bugs associated with padding handling in: Conformer Convolutions (1), Initial Subsampling (2), and Positional Encodings (3).

Conformer Convolutions (1) The depthwise and pointwise convolutions of the Conformer convolution module do not consider the presence of padding and produce a non-padded output with non-zero values adjacent to the input sample. These values modify the behavior of the subsequent batch normalization and of the other convolutions, leading to incorrect alterations of the valid values.

<table>
<thead>
<tr>
<th>P₀₀</th>
<th>P₀₁</th>
<th>P₀₂</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₂₀</td>
<td>P₂₁</td>
<td>P₂₂</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(1) Before shifting, the Relative PE matrix \((P₀₀, ..., P₂₂)\) is padded (zero values).

<table>
<thead>
<tr>
<th>0</th>
<th>P₀₀</th>
<th>P₀₁</th>
<th>P₀₂</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(2) When relative shift is applied to the Relative PE matrix without considering padding, some values of the padding area (in red) are incorrectly moved to the non-padding area.

<table>
<thead>
<tr>
<th>0</th>
<th>P₀₀</th>
<th>P₀₁</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀₀</td>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(3) When relative shift is applied to the Relative PE matrix considering padding, the values \(P₀₀, ..., P₂₂\) are not moved to the padding area.

---

8https://github.com/sooftware/conformer
9We emphasize that our intention is not to single out the shortcomings of individual libraries. Conversely, we are extremely thankful for the invaluable contribution they represent to our community. Our analysis is only intended to further improve the reliability of codes and, consequently, of the experimental results, which we believe is of utmost importance.

3660
Table 2: Bugs present in the analyzed repositories. NA stands for “Not Applicable”.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fairseq-ST</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ESPnet-ST</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NeMo</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SpeechBrain</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Conformer</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TorchAudio</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Initial Subsampling (§2) The two initial convolutions that subsample the input sequence by a factor of 4 do not consider padding. Hence, the second convolution is fed with non-zero values adjacent to the input sequence, which are wrongly considered in the computation of the last valid elements.

Positional Encodings (§3) The relative sinusoidal positional encodings (PES), which are added to the attention matrix, are computed by shifting a sinusoidal matrix. This shifting operation first prepends a zero column to the sinusoidal matrix and then reshapfs it so that the last element of the first row becomes the first element of the second row, the last two elements of the second row become the first ones of the third row, and so on. By doing this, this operation assumes that all elements are valid. However, when a sequence is padded, only part of the attention matrix is valid (in green in Figure 1.1) and spurious values are moved to the beginning of the next row (Figure 1.2). In Figure 1, for the sake of clarity of the example, we set to 0 the PE in the padding area. While this is not what happens in practice (as the padding area contains other sinusoidal PEs), it shows that the correct values are discarded and the final matrix significantly differs from the one obtained without padding, which is instead shown in Figure 1.3.

In Table 2, we report the presence (or absence) of these bugs for each analyzed codebase in its current version. All the implementations but one (NeMo) are affected by §1. Also, all are affected by §2 and §3, except for TorchAudio, whose implementation neither includes relative positional encodings in the attention nor the initial sub-sampling convolutional layers. Having ascertained that all the analyzed implementations contain at least one bug, the next sections will concentrate on their impact on ASR and ST results and, in turn, the related findings.

4.2 Experimental Settings
We train and evaluate ASR and ST models on MuST-C v1.0 (Cattoni et al., 2021), which contains parallel speech-to-text data with English (en) as source language and 8 target text languages, namely Dutch (nl), French (fr), German (de), Italian (it), Portuguese (pt), Romanian (ro), Russian (ru), and Spanish (es). For ASR, we use the en-es section (the largest of the corpus). For ST, 8 different models are trained, one for each language direction. Evaluation is performed on the tst-COMMON, by computing word error rate (WER) for ASR and BLEU with SacreBLEU (Post, 2018) for ST. We assess statistical significance using bootstrap resampling (Koehn, 2004) with 95% confidence interval. Detailed experimental settings are reported in Appendix B.

Trainings and inferences were performed on, respectively, two and one A40 GPU(s). On Ampere GPUs, PyTorch computes convolutions and matrix multiplications with TensorFloat-32 (TF32) tensor cores by default. TF32 speeds up the computation but introduces numeric errors that can cause small random fluctuations, e.g. in the presence of padding. In the following, we experiment both with and without TF32 (both at training and inference time) because padding has no effect on the final outputs only when TF32 is disabled.

4.3 Impact of the Identified Bugs
We evaluate the impact of the identified bugs (§4.1) on ASR and ST results by varying the IBSs as increasing the batch size introduces more padding, amplyfying the effects of the bugs. Initially, experiments are conducted on our correct codebase (§2). Subsequently, we enable single precision (TF32). Then, we reintroduce the bugs individually (§1, §2, and §3), and all together (§1, §2, and §3).

ASR Table 3 shows, in comparison to §2, the impact of TF32 and of the different bugs on ASR performance. First, TF32 causes a not statistically significant quality drop (+0.21 WER), which does not vary with the IBS (despite the presence of minor variations in the outputs attested by a slightly different number of generated words). When the bugs are present (§1, §2, and §3), instead, the performance becomes sensitive to the IBS. This is particularly evident with §1, which significantly increases
4.4 Impact of Building on Incorrect Code

We now showcase how incorrect code can lead to misleading conclusions when experimenting with new techniques. We choose to evaluate the CTC compression (Liu et al., 2020; Gaido et al., 2021) because we speculate that it limits the negative effects of the bugs identified in §4.1, as it reduces the sequence lengths and, in turn, the amount of padding. Introduced in the context of Transformer-based models, CTC compression reduces training and inference times, as well as VRAM requirements, while yielding minimal (not statistically significant) gains in terms of translation quality (Gaido et al., 2021). A detailed description of the CTC compression is provided in Appendix C.

ASR Table 6 shows the effects on ASR performance of introducing CTC compression (CTC Compr.) into the codebase with all bugs (shade) and without them ( ). CTC compression causes a small and not statistically significant performance degradation (+0.12 WER) when the correct implementation ( ) is used (in accordance with the findings on the Transformer architecture). When bugs are present in the codebase (shade), instead, the outcome is overturned: CTC compression brings statistically significant gains (-0.93 WER even with 1 as IBS). These observations lead to the conclusion that building on incorrect code can produce misleading conclusions when experimenting with new techniques.
<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
<th>IBS</th>
<th>en-de</th>
<th>en-es</th>
<th>en-fr</th>
<th>en-it</th>
<th>en-nl</th>
<th>en-pt</th>
<th>en-ro</th>
<th>en-ru</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformer</td>
<td></td>
<td>1</td>
<td>24.67</td>
<td>30.34</td>
<td>36.22</td>
<td>25.73</td>
<td>30.04</td>
<td>30.55</td>
<td>23.43</td>
<td>17.29</td>
<td>27.28</td>
</tr>
<tr>
<td>+ CTC Compr.</td>
<td>✓</td>
<td>10</td>
<td>24.97</td>
<td>30.48</td>
<td>36.43</td>
<td>26.25*</td>
<td>30.31</td>
<td>30.09†</td>
<td>24.67*</td>
<td>17.35</td>
<td>27.28</td>
</tr>
<tr>
<td>Conformer</td>
<td></td>
<td>1</td>
<td>24.68</td>
<td>28.57</td>
<td>35.70</td>
<td>25.81</td>
<td>29.68</td>
<td>30.22</td>
<td>23.52</td>
<td>15.83</td>
<td>26.69</td>
</tr>
<tr>
<td>+ CTC Compr.</td>
<td>✓</td>
<td>10</td>
<td>25.21*</td>
<td>30.72*</td>
<td>36.18*</td>
<td>26.01</td>
<td>29.35*</td>
<td>30.14</td>
<td>23.95*</td>
<td>17.06*</td>
<td>27.36</td>
</tr>
</tbody>
</table>

Table 6: WER for ASR of the correct/incorrect codebase with and without CTC Compr. as IBS varies (1, 10, and 100). *† indicate that the improvement/degradation of CTC Compr. is statistically significant.

<table>
<thead>
<tr>
<th>Code</th>
<th>Model</th>
<th>IBS</th>
<th>en-de</th>
<th>en-es</th>
<th>en-fr</th>
<th>en-it</th>
<th>en-nl</th>
<th>en-pt</th>
<th>en-ro</th>
<th>en-ru</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Conformer</td>
<td></td>
<td>10.52</td>
<td>10.52</td>
<td>10.52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>+ CTC Compr.</td>
<td>✓</td>
<td>10.52</td>
<td>10.52</td>
<td>10.52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>Conformer</td>
<td></td>
<td>11.32</td>
<td>14.25</td>
<td>54.56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>+ CTC Compr.</td>
<td>✓</td>
<td>10.39*</td>
<td>10.34*</td>
<td>10.81*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7: BLEU for ST of the correct/incorrect codebase with and without CTC Compr. as IBS varies (1, 10, and 100). *† indicate that the improvement/degradation of CTC Compr. is statistically significant.

misleading findings. Besides, the best overall result is achieved with the ✓1,2,3 codebase (with 10 as IBS and CTC compression), reiterating that high scores do not imply code correctness.

5 Increasing Research Code Correctness

After demonstrating that the current tendency to assess code correctness solely based on the reported results (§3) potentially leads to wrong findings (§4), in this section we propose countermeasures. Specifically, we aim at fostering the adoption of SQA best practices in two ways: 1) by releasing a Python package (pangoliNN) for testing neural networks and assisting researchers in the verification of code correctness; 2) by proposing the integration of current conference checklists with recommendations for SQA best practices (§5.2).

5.1 pangoliNN

As discussed in §2, testing software is the only way to enforce that it works correctly. Therefore, as the first and foremost method to increase code correctness, we recommend the extensive implementation and adoption of Unit Tests (UTs) to check that the code has the expected behavior (Liskov, 1975; Goodenough and Gerhart, 1975; Huizinga and Kolawa, 2007; Kassab et al., 2017). Ideally, this should be done prior to writing the actual code, following the so-called “test-driven development practice” (Beck, 2002). UTs should cover all the assumptions about how the code works (e.g., ensuring that the presence of padding does not alter the results). While achieving complete test coverage is a utopian objective, the higher the coverage, the higher the quality of the codebase.
To ease this work, we introduce pangoliNN, a Python package specifically designed for testing neural modules. Built upon the widely used PyTorch library (Paszke et al., 2019), pangoliNN offers a collection of pre-defined tests that enforce specific behaviors of the modules. Its primary objective is to simplify and expedite the process of testing neural networks, alleviating researchers from the burden of creating UTs from scratch. Indeed, writing UTs may initially be perceived as an additional and undesirable cost, although ample literature dispels this perception. Williams et al. (2003), for instance, proved that the inclusion of UTs does not hamper code-writing productivity, Ellis et al. (2004, 2006) showed that the perceived cost is “exaggerated”, and Hevery (2009) that the initial overhead pays off by saving time spent on manual experiments.

Furthermore, unlike manual experiments that are often resource-intensive and environmentally impactful (Strubell et al., 2019), UTs are generally lightweight (e.g., they do not involve any training phase). As such, writing UTs would contribute to the environmental sustainability of NLP research, facilitating the transition to Green AI (Schwartz et al., 2020). Also, UTs do not require any pre-trained model to run, as their nature and goal greatly differ from assessing the quality of a trained model, as recently proposed with behavioral testing (Ribeiro et al., 2020). Through behavioral testing, we check whether a specific instance properly handles different aspects, such as linguistic phenomena (e.g., negation, co-references), and/or produces correct outputs with challenging inputs. Through UTs, instead, we assess the behavior and robustness of the code itself (rather than of model instances), by verifying whether assumptions about properties of the network or about its behavior in specific conditions (e.g., not being influenced by the presence of padding) are respected.

Currently, pangoliNN includes tests for two aspects: i) proper handling of padding and batching, ensuring consistent output of neural modules irrespective of padding presence; and ii) addressing causality by verifying the independence of module output from future elements in the input sequence, which is crucial for autoregressive and other sequence-to-sequence models. It also features comprehensive documentation with simple examples to guide researchers in its usage. Moreover, pangoliNN itself is extensively unit tested, and these UTs provide additional implicit guidance on how to effectively utilize the package.

Despite being in its initial stage (for the limited number of tests currently covered), we argue that the first release of pangoliNN represents a milestone toward increasing the quality and trustworthiness of NLP research code and, in turn, outcomes. We hope that it will be embraced and expanded upon by the research community, with the integration of additional tests, ultimately growing it into a comprehensive testing library for neural networks.

5.2 Code-quality Checklist

As a complementary initiative, we also propose to integrate the existing conference checklists with questions targeting the improvement of code correctness and quality in research (Table 8). It is worth mentioning that we have strictly adhered to these guidelines throughout the development of both pangoliNN and of our padding-safe implementation of the Conformer architecture.

The first questions (Q1-2) focus on the adoption of UTs (possibly leveraging pangoliNN), whose importance has been stressed in the previous section. However, the presence of UTs alone does not guarantee that the code works. Indeed, UTs should be executed every time the codebase is modified, even in case of a seemingly unrelated change, as the validity of a test expires whenever the software is edited (as seen in §2). This is commonly enforced through continuous integration (CI), which executes all UTs at every code change (Duvall et al., 2007). A running and successful CI offers the supplementary advantage of providing implicit guidance on the installation and execution of the code for individuals attempting to replicate a study. Furthermore, it mitigates the occurrence of replication failures due to syntax or runtime errors, as it often happens in current NLP artifacts (Arvan et al., 2022). Such failures can arise because the released version of the code might slightly differ from those used for the experiments due to small refactorings prior to the release. For this reason, Q3 and Q4 respectively focus on test execution and on the presence of a CI, so as to ensure that the checks of the UTs are actually respected.

Lastly, we encourage (Q5) the adoption of a code reviewing practice (Baum et al., 2016), in which

---

12As a pangolin looks for bugs and catches them, this library aims at finding bugs in neural networks (NN). Hence the name.

13Estimated in 16%-35% of the overall software development cost (George and Williams, 2003; Hevery, 2009).
1. Have you tested your code with relevant tests?
2. Have you tested assumptions about code behavior with Unit Tests (UTs)?
3. Have UTs been executed on the code version used for the experiments and, if applicable, the publicly released version?
4. Does the repository contain a continuous integration that runs the UTs?
5. Has every contribution to the codebase been reviewed by at least one person?

Table 8: Code-quality Checklist.

all changes are reviewed and approved by a person different from the code author.\textsuperscript{15} Code review is a lightweight and informal process compared to code inspection (Fagan, 1976) and has been shown to cause little overhead for most code changes (Sadkowski et al., 2018). It consists in reading and commenting on the source code for a change, which should be kept small and focused on one single aspect or new feature. This aims not only at avoiding bugs, but also at improving code readability and documentation (Baum et al., 2017; Chen et al., 2019; Bahaidarah et al., 2022; Trisovic et al., 2022), and, in turn, reusability and reproducibility. In addition, it serves as a powerful tool for knowledge transfer (Bacchelli and Bird, 2013), thus novices would particularly benefit from it.

As a final note, we would like to emphasize that our proposed “Code-quality Checklist” should be interpreted as the “Reproducibility Checklist” now required by many top-tier venues: though strongly encouraged, following the checklist is not mandatory for paper submissions and its intent is fostering software quality and correctness rather than certifying it. Specifically, it will encourage awareness of SQA concepts and coding best practices, especially among researchers who have not been exposed to them during their education.

6 Conclusions

In parallel with the current efforts to enhance the reproducibility of NLP research, this paper urges similar actions targeting the improvement of research software quality, underscoring its importance for the reliability of research findings. In comparison to the attention given to reproducibility within our community, we observed the predominant neglect of code correctness and elaborated on the risks associated with assessing soundness solely on the basis of experimental results. As we empirically demonstrated through a case study involving the widespread Conformer architecture, such risks include the potential of drawing misleading conclusions from positive results obtained using flawed code. As a countermeasure, besides releasing a corrected Conformer implementation, we created the \texttt{pangolinn} Python package to facilitate testing neural models and proposed the adoption of a “Code-quality Checklist” aimed at fostering coding best practices. While we acknowledge that these solutions are not a panacea, their purpose is to raise awareness within the NLP community about the importance of software quality. We hope that our endeavor will inspire a collective commitment to developing high-quality and reliable code.

Acknowledgements

We acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU. We acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support.

Limitations

To back up our call to action toward the adoption of coding best practices aimed at fostering correctness and improving the quality of the developed software, we presented a case study involving the use of the Conformer architecture in the two most popular speech processing tasks: speech recognition and translation. Although the effects of the presence of bugs might be found also in other scenarios, such as text-to-speech, speech emotion recognition, spoken language understanding, and speech separation, we did not cover them in this paper. While the undesired effect of the bugs we isolated (and corrected) was empirically demonstrated, extending the analysis to other research areas would be a natural extension of our study, which could provide a more comprehensive understanding of the impact of the identified bugs on the broader NLP community working on speech-related tasks.

Moreover, in our case study, we examined the open-source implementations of Conformer, in which we identified three types of bugs related
to the Convolution Module, Initial Subsampling, and Positional Encodings. While we found efficient solutions for the first two bugs, for the last one our fix introduces a significant overhead. As a result, the implementation we release, although correct, increases the training time of the models. We are confident that, by open-sourcing our code, the community will soon find a way to optimize it and overcome this limitation, capitalizing on our findings and spreading the use of more reliable versions of a state-of-the-art architecture.

References


Layan Bahaidarah, Ethan Hung, Andreas F. De Melo Oliveira, Jyotsna Penumaka, Lukas Rosario, and Ana Trisovic. 2022. Toward reusable science with readable code and reproducibility. In 2022 IEEE 18th International Conference on e-Science (e-Science), pages 437–439, Los Alamitos, CA, USA.


Xiaoli Chen, Sunje Dallmeier-Tiessen, Robin Dasler, Sebastian Feger, Pamfilos Fokianos, Jose Benito Gonzalez, Harri Hirvonsalo, Dinos Kousidis, Artemis Lavasa, Salvatore Mele, Diego Rodriguez Rodriguez, Tibor Šimko, Tim Smith, Ana Trisovic, Anna Trzcinska, Ioannis Tsnaktsidis, Markus Zimmermann, Kyle Cranmer, Lukas Heinrich, Gordon Watts,


Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for activation functions.


A Conformer in ASR and ST

ASR is the task in which an audio containing speech content is transcribed in its original language. In ST, instead, the source audio is translated into text in a different language. Nowadays,
both tasks are commonly performed with end-to-end (or direct) models (Graves and Jaitly, 2014; Chorowski et al., 2014; Bérand et al., 2016; Weiss et al., 2017), whose architecture is based on the Transformer (Vaswani et al., 2017). The Transformer has been adapted to work with audio inputs (Dong et al., 2018; Di Gangi et al., 2019) by introducing two convolutional layers that shrink the length of the input sequence by a factor of 4, so as to reduce the otherwise excessive memory requirements. More recently, Gulati et al. (2020) proposed the Conformer: a novel architecture with a modified encoder that led to significant improvements in both ASR and ST (Inaguma et al., 2021).

The changes introduced in the Conformer encoder layer structure can be summarized as follows: i) relative sinusoidal positional encodings (Dai et al., 2019) are introduced in the self-attention for improved generalization with respect to varying input lengths; ii) the FFN sublayer is replaced by two FFNs that wrap the self-attention, inspired by the Macaron-Net (Lu et al., 2019); iii) a convolution module (Figure 2) is added after the self-attention, before the second FFN layer. The convolution module, which is wrapped in a residual connection, applies layer normalization, followed by a pointwise convolution that doubles the dimension of the feature vector, which is restored to its original size by a Gated Linear Unit (GLU) activation function (Dauphin et al., 2017). Then, a depthwise convolution with 31 kernel size is applied before a batch normalization (Ioffe and Szegedy, 2015), followed by the Swish activation function (Ramachandran et al., 2017), and another pointwise convolution. Lastly, a dropout module (Srivastava et al., 2014) randomly zeroes out a percentage of the features to prevent the network from overfitting.

### B Experimental Settings

Our Conformer-based architecture is composed of 12 Conformer (Gulati et al., 2020) encoder layers and 6 Transformer (Vaswani et al., 2017) decoder layers, with 8 attention heads each. Embedding size is set to 512 and hidden neurons in the feed-forward layers to 2,048, with a total of 114,894,730 model parameters. Dropout is set to 0.1 for feed-forward, attention, and convolution layers. The kernel size of the Convolution Module is set to 31 for both point-wise and depthwise convolutions. We train all the models using Adam (Kingma and Ba, 2015) optimizer (betas (0.9, 0.98)) and label-smoothed cross-entropy (LSCE) loss (smoothing factor 0.1). We also use an auxiliary Connectionist Temporal Classification or CTC loss (Graves et al., 2006) during training to ease convergence and obtain competitive results without pre-training the encoder with that of an ASR model (Gaido et al., 2022). The auxiliary loss is summed to the LSCE with 0.5 relative weight. The learning rate is set to $2 \cdot 10^{-3}$ with Noam scheduler (Vaswani et al., 2017) and 25k warm-up steps. The vocabularies are based on SentencePiece models (Kudo and Richardson, 2018) with size 5,000 (Inaguma et al., 2020) for the English source and 8,000 (Di Gangi et al., 2020) for the ST target languages. We set 100k maximum updates with early stopping after 10 epochs without loss decrease on the dev set and average 5 checkpoints around the best (best, two preceding, and two following). All trainings are performed with 40k tokens as batch size and 4 as update frequency on two GPUs. All other settings are the default of Fairseq-ST (Wang et al., 2020), which we forked as a base of our implementation. SpecAugment (Park et al., 2019) is applied during training, while utterance-level Cepstral mean and variance normalization is performed both at training and inference time. Trainings lasted 18-33 hours depending on the model configuration (e.g., with or without the fixes) and the language pair due to the different sizes of the training data.

### C CTC compression

CTC compression has been proposed to reduce the difference in terms of sequence length between corresponding audio and text representations. In contrast with fixed reduction methods like max pooling or strided convolutions that apply a pre-determined reduction to each sequence, CTC compression leverages the probability distribution over the source vocabulary augmented with a `<blank>` symbol produced by the CTC module. These probabilities are used to assign a label (the most likely
one) to each vector of the sequence and collapse contiguous vectors corresponding to the same label by averaging them. By dynamically determining which vectors of the audio sequence should be merged, it tries to avoid the mismatch in terms of sequence length with the sub-word sequence of the corresponding transcript.