A Guide to Computational Reproducibility in Signal Processing and Machine Learning

Joseph Shenouda and Waheed U. Bajwa

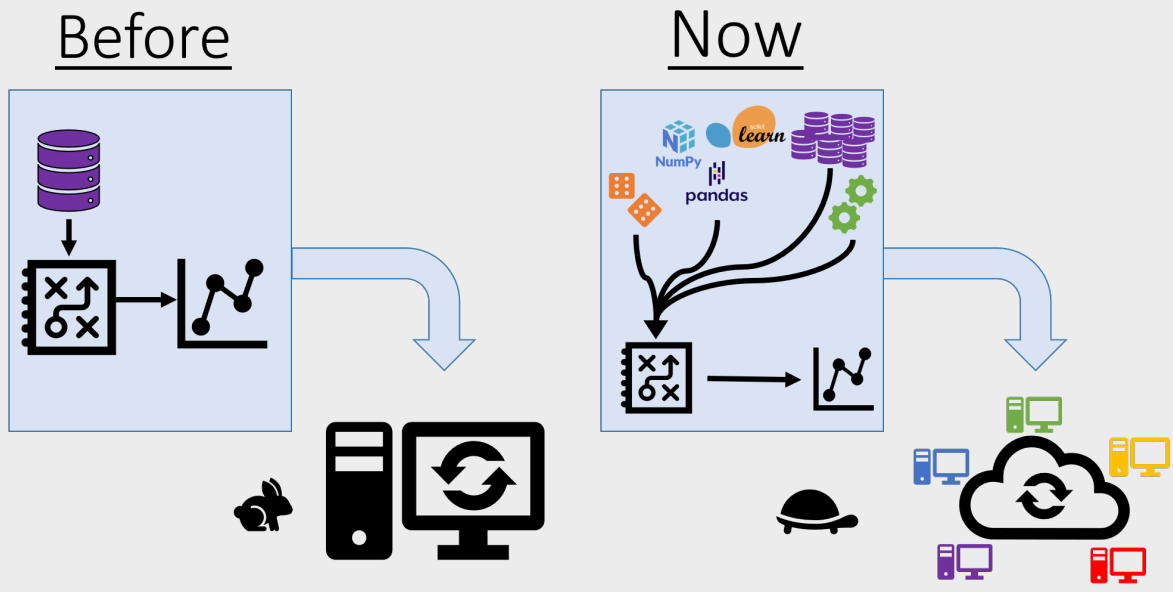
I. Introduction

A computational experiment is deemed *reproducible* if the same data, methods, and implementations are available to replicate quantitative results by any independent researcher, anywhere and at any time, granted they have the required computing power. Such *computational reproducibility* is a growing challenge that has been extensively studied among computational researchers as well as within the signal processing and machine learning research community [1], [2]. Note, however, that there exists much confusion about how one defines *reproducibility* and how it differs from similar terms such as *repeatibility* and *replicability* [3], [4]. We follow the approach in [5], which considers reproducibility to exist on a spectrum, and view computational reproducibility as a minimum standard for assessing the scientific impact of a study in signal processing and machine learning. Other works have considered this notion of reproducibility under the term 'replicability' [6] or 'Results Reproducibility' [4]. This differs from other interpretations of reproducibility such as 'Inferential Reproducibility' where qualitatively similar results can be obtained by an independent replication of a scientific study [4], which may help validate the meaningfulness and statistical significance of the results and conclusions. While computational reproducibility is only one component within the broader spectrum of reproducibility, it is nonetheless an important one that computational researchers are still struggling to solve [7]–[9]. Moreover, it is only through first achieving computational reproducibility that one can begin to address the other components of reproducible research.

Signal processing research is becoming increasingly reliant on computational experiments to test hypotheses and validate claims, which is in contrast to the yesteryears when one typically used computational experiments to elucidate rigorous theory and mathematical proofs. Therefore it has become more important than ever to ensure the reproducibility of computational experiments, as this is the first step in confirming the validity of research claims supported through the outcomes of computational experiments. But this is not turning out to be an easy task. The paradigm shift from the theory-driven research to the compute-driven claims in signal processing and machine learning has been facilitated by powerful computing resources, accessibility of massive datasets, and a myriad of new libraries and frameworks (such as NumPy

[10], Scikit-learn [11], MATLAB Toolboxes [12], and TensorFlow [13]) that provide a layer of abstraction allowing for rapid implementation of complex algorithms. Unfortunately this changing research landscape is also bringing with it new obstacles and unseen challenges in developing reproducible experiments.

Computational experiments today often incorporate various scripts for preprocessing data, running algorithms, and plotting results, all while utilizing huge datasets which require computing clusters that often take days or weeks to finish computing with multiple manual interventions needed to successfully produce the desired results. This is contrary to the way computational experiments used to be conducted and the way new researchers are introduced to computational resources in the classroom, where they typically use simple and intuitive interactive computing software consisting of a single script that runs locally on one's computer [14]. This new paradigm of computational experiments is now requiring the scientific community to rethink how we publicize and share our code to encapsulate all the necessary information about our experiments and make computational reproducibility practically possible. Additionally our extensive dependence on libraries and frameworks leads to brittle codebases that typically only output correct results when executed on the original machine of the researcher. Furthermore, the nature of these data-driven experiments often require careful parameter tuning, random number generations, and data preprocessing, all of which are independent from the main finding, such as a new algorithm, being implemented or investigated.



Evolution of Computational Experiments: Computational experiments in signal processing and machine learning today have transformed from the way they used to be conducted in the early days of these

fields. Due to the rise in the availability of computational resources and large datasets, many of our computational experiments can no longer be carried out on local workstations, as was the norm in the past. Rather, they are often carried out on large computing clusters and can take hours or days to complete. This fact, coupled with the dependencies on multiple third-party libraries, hyperparameter tuning, and random number generators makes it very time consuming, and sometimes nearly impossible, to try and reproduce published computational results by trial and error.

Due to these new challenges most experiments have become difficult, if not impossible, to be reproduced by an independent researcher. As an anecdote, when attempting to reproduce computational results in our lab from a paper published just months prior, even the original authors of the experiment were unable to completely reproduce the results. However, this phenomenon is not unique to our lab. In 2016 a survey conducted by the journal Nature found that 50% of researchers were unable to reproduce their own experiments [15]. And while the issue of reproducibility has been discussed in the literature [1] and specifically within the signal processing community [16], [17], it is still unclear to most researchers what are the most practical approaches to ensure *computational* reproducibility without impinging on their primary responsibility of conducting research. This is because the guidelines and best practices provided for computational reproducibility in the existing works [16], [17] do not account for all the obstacles to reproducibility of the increasingly complex and large-scale computational experiments. In addition to the complexity of modern computational experiments, these obstacles include the potential for human errors and the rapidly evolving technological landscape that is changing at an unprecedented rate. This article complements the existing works by explicitly focusing on these and related obstacles for computational reproducibility and, in contrast to the discussion in [16], [17], advocates that researchers should plan for the computational reproducibility of their experiments long before any code is written.

In summary, although works such as [1], [16], [17] have helped researchers understand the importance of making computational experiments reproducible, the lack of a clear set of standards and tools makes it difficult to incorporate good reproducibility practices in most labs. It is in this regard that we aim to present signal processing researchers with a set of practical tools and strategies that can help mitigate many of the obstacles to producing reproducible computational experiments.

A. Why Computational Reproducibility

Making computational experiments reproducible is a necessary step for ensuring the credibility of the conclusions made from a research study. If researchers are regularly unable to validate the computational results in a study, it becomes impossible to investigate whether or not the latest results presented in a

research paper are indeed state-of-the-art. For example, a group of researchers recently published work that presented a new recurrent neural network architecture for language modelling which appeared to achieve state-of-the-art performance in terms of *perplexity* on the Penn Treebank dataset. However, a different group of researchers found that they can control the hyperparameters in a more fine-tuned way than the original researchers. This led to the discovery that—after careful hyperparameter tuning—the traditional long short-term memory (LSTM) recurrent neural network model can achieve better perplexity on the same dataset, contrary to what was found by the first group. This specific example demonstrates the significance of hyperparameters and other meta data in assessing the performance of new machine learning algorithms. The claims of state-of-the-art performance in particular are directly related to these incidental details and without the ability to reproduce previous computational results it can become difficult to discern the true novelty of new research findings.

This example, which is far from a rarity, illustrates the broader point that the field of computational sciences cannot truly advance until we are confident in the past progress. Not only does ensuring computational reproducibility protect the integrity of the research, it also allows fellow researchers to develop their own experiments quickly by utilizing code written by others investigating the same problem. In addition, implementing reproducible computational experiments has numerous benefits for the researchers themselves [18]. By utilizing the techniques outlined in this article and making the experiments as transparent and detailed as possible, one can avoid catastrophes such as a hard drive that gets corrupted, resulting in a loss of all data and source code, or a loss of older promising results due to a bug that was introduced in the code later on. Additionally, old research problems often get revisited and improved upon; thus having reproducible code for the experiments in the original study can save hours of frustration for the researcher(s) trying to reproduce old computational results, before embarking on the new research. Finally, making one's computational experiments reproducible can add value to the research itself; by lowering the barrier of entry for other researchers to engage and expand on the experiments, it makes the research more accessible to the community, which can in turn lead to more impactful work [19].

While the importance of reproducible research in general and computational reproducibility in particular is obvious to many, the exact tools and techniques that one must utilize when building computational experiments to ensure that they are reproducibile for the foreseeable future after publication are still not very clear. In particular, a common misconception is that simply publishing all the code and data used to obtain the results makes computational experiments reproducible. But this is almost never the case, as the raw source code and data along with the paper alone cannot give enough details on how to run

the experiment, the necessary dependencies, or the required computational power. It is in this regard that we worked on making two experiments from two different research projects here at the INSPIRE Lab reproducible and, in the process, investigated the best ways to create reproducible experiments organically, taking into consideration the extra overhead that comes with such an endeavour. The best techniques and tools were considered in light of the fact that one's goal as a computational researcher is typically to conduct and disseminate research and not maintain or develop commercial software. By sharing our experiences and best techniques with the readers of the IEEE Signal Processing Magazine, it is our hope that we can go beyond just highlighting the importance of computational reproducibility by providing a clear and practical guide for developing experiments in a reproducible manner.

We have organized the rest of the article as follows. First, the common pitfalls to computational reproducibility are explained. Next, the standards for computational reproducibility are discussed. The last three sections discuss solutions and tools that can be utilized to avoid the reproducibility pitfalls discussed earlier. These topics include version control for organizing a dynamic and collaborative codebase, package managers for handling multiple dependencies and finally, techniques on how to eventually share the code accompanying the computational experiments.

II. The Computational Reproducibility Pitfalls

Most papers in the field of signal processing and machine learning tend to include a section at the end where the authors explain their computational experiments along with figures that provide some evidence that the proposed research has practical implications. However, due to the limitations in space and in the interest of conciseness, this section of the research papers cannot provide all the necessary details to reproduce the results. Even when computational experiments are only meant to give some justification for a rigorous theory, these results are important and should be explained as clearly as the theorems and proofs on which they are based. While publishing all the source code used for the computational experiments is a step in the right direction and may seem to fill in all the missing gaps from the research paper, it is often still not enough to completely reproduce the original results [3]. This is in part due to the fact that most researchers are not trained on how to write clean, maintainable code in a collaborative setting [20]. This lack of training has the potential to result in highly disorganized code that is difficult to parse and understand by an independent researcher.

Another potential pitfall we identified, which can seem benign for small experiments, is that researchers have multiple versions of the same computational experiment with slightly different code, which makes it impossible to know which was the one used in obtaining the reported results. Thus, attempting to reproduce the final set of results requires one to first run each version of the codebase individually until

they produce the desired results. On large computational experiments that take days to finish, this is of course impractical. Another issue is that researchers typically do not describe in enough detail the dependencies needed for running the code used for the experiment [3]. Even when the dependencies are mentioned, researchers might omit the exact version that was used when originally obtaining results, which could make computational reproducibility impossible for those attempting to run the code on a different machine. For example, if the original code used a certain feature from a library that has since been deprecated in later versions, those running the code using the latest version of the library cannot reproduce the results, and may not know that they need to downgrade their dependency. Therefore, there must be a way for the original researchers to preserve and share the exact computational environment used when generating results in order to share it with others looking to run their code to reproduce results. Additionally, as mentioned earlier, the source code alone does not provide instructions on how to run the computational experiments nor the order in which the scripts should be executed. Another piece of information that is rarely mentioned in enough detail when sharing the code is the computational power needed for an experiment [3] and the time it takes to finish executing. This information is necessary for those trying to reproduce the computational results as they must first verify that they are equipped with the right amount of computational power to run the code.

Even when these pitfalls are accounted for, there is still the challenge of sharing the necessary meta data accompanying the experiments. The description of the experiments present in most research papers simply cannot encapsulate all the necessary meta data needed to successfully reproduce computational results. For instance, while the authors typically mention the dataset being used in a particular experiment, there may exist some ambiguity about the exact source of the data. And this can be true even for established benchmark datasets such as the “House” image or the MNIST dataset [21]. While the authors might believe that these are “standard” datasets, it is often the case that different versions of these datasets are circulating around the internet, each with slight variations that may not be immediately noticeable but can yield different results when used for the same experiment [16]. Even if the sources of the datasets are explained, the preprocessing steps performed on the data can be vital to obtaining the published computational results, but these might not be thoroughly explained in research papers. All of these seemingly benign or superfluous details can have an effect on the computational results produced from the experiment that others may be trying to reproduce. Moreover, for computational experiments that use synthetic data, the way in which the data was produced may not be explained in enough detail. Furthermore, signal processing and machine learning algorithms typically rely on finely tuned hyperparameters that, when changed, can also give different results and even invalidate

the conclusions made in the paper, yet the hyperparameters or the methods by which they are found are not always described in enough detail [3]. For example, most machine learning experiments make use of cross-validation to find optimal hyperparameters and while the researchers may mention this detail, they might omit information about how exactly the dataset was split up. This in turn could lead to different hyperparameters when the experiment is run by someone trying to reproduce the original results.

Finally, most experiments rely on random number generators somewhere in the codebase. An example of this would be the stochastic gradient descent (SGD) algorithm [22], which is a popular method for large-scale training in machine learning. In each iteration of the vanilla SGD, a random sample from the dataset is used to compute the gradient of the loss function and update the parameters of interest. As a result, the rate of convergence and the values of the optimized parameters depend on the order in which the samples were selected [23] and will be different if the randomly chosen sample in each iteration is not consistent every time the experiment is run. By not saving the random seed in the code as part of the experimental meta data, the sequence of randomly chosen samples will vary every time someone tries to run the experiment, making it almost impossible to exactly reproduce the original computational results each time the codebase is run.

In the rest of this article we discuss what it means for a computational experiment to be computationally reproducible and the potential techniques for overcoming each of the pitfalls discussed. While the suggested methodologies for creating computational experiments may incur some additional work for the researcher, it strikes a good balance between ensuring reproducible results and becoming an obstacle to further research. Furthermore, through practice over time, it is our hope that the tools and techniques discussed below will become commonplace for researchers, giving them the ability to naturally create computional experiments that are readily computationally reproducible.

III. The Goals for Reproducibility

There has been a lot of discussion across many domains, including within the signal processing and machine learning community, about how to successfully make computational experiments reproducible. However, much of the work discussing computational reproducibility tends to advocate for new software tools that can be used to easily publish computational experiments that are reproducible. One such example is the "Whole Tale" [24], a platform that enables researchers to create *tales*—executable research objects that capture data, code, and computational environments—for the computational reproducibility of the research findings. While tools like this may seem promising, they have limitations. The first is that these reproducibility tools attempt to encapsulate the whole process of running an experiment from preprocessing to plotting, treating the computational experiment as a blackbox and this in turn

TABLE I

A TABLE OF THE COMMON PITFALLS DISCUSSED IN THIS SECTION AND WHERE TO FIND THEIR RESPECTIVE REMEDIES THROUGHOUT THIS ARTICLE.

Pitfall	Short description	Remedy
1	Disorganized codebases and multiple versions.	Section IV
2	Differences in computational environments and failure to disseminate necessary dependencies.	Section V
3	Missing critical meta data such as hyperparameters, computing power and dataset sources.	Section VI

leaves the actual code in a state that is difficult to interpret for those who might be interested in digging deeper or expanding upon it. Additionally, due to the fact that they attempt to encapsulate every component of an experiment, they tend to be highly inflexibility and may not be appropriate for every computational experiment, making the process of creating computationally reproducible experiments even more cumbersome. This is especially true for the research labs that tend to focus more on the theoretical and algorithmic aspects of signal processing and machine learning, rather than the applied aspects, and therefore provide only simple computational experiments to give empirical justifications for their claims. In such labs the actual codebases for the computational experiments are not very large and they tend to focus narrowly on a newly proposed idea, insight, or algorithm, which is in contrast to large data analysis projects found in other computational sciences. In addition, the encapsulation-based reproducibility tools are often funded by grant money and are typically only maintained by a single lab; thus when the grant money runs out, there is no guarantee that the tool will be maintained and it may become obsolete within a few short years [25]. Therefore relying on these computational reproducibility tools would require one to relearn a new platform for publishing their experiments in a computationally reproducible way every time an old one gets abandoned. In summary, while an encapsulation tool could in theory be the optimal solution to the computational reproducibility crisis, there currently does not exist any widely adopted off-the-shelf tool that overcomes the issues highlighted here.

Because of the aforementioned reasons, the focus of this article is not on finding or creating the best computational reproducibility software that can automatically encapsulate all the components of the computational experiment. Rather, the goal is to find how one can achieve computational reproducibility

by relying on robust open-source tools, used across both industry and academia. This involves formulating a methodology for developing computational experiments such that the reproducible codebase supporting a new research finding can be released in parallel with the publication of the paper. To be more precise, the methodology should enable any independent researcher studying a similar problem to obtain, understand, and easily run the code used in the computational experiments in order to reproduce the exact same figures, plots or values without enduring a painstaking trial and error process. Furthermore, this should be possible without the need to ever contact the researchers responsible for the original computational experiment.

The goals of computational reproducibility espoused in this article also emphasize the importance of making experiments computationally reproducible throughout their development, since trying to retroactively make the experiments computationally reproducible after the results have been obtained and published is usually a much more difficult task. If computational reproducibility is only attempted after publication, the researcher is likely to have already fallen into one of the pitfalls mentioned earlier, and must struggle with first reproducing their own results, perhaps unsuccessfully, before being able to share the code with others. While it may be cumbersome at first, after researchers get accustomed to the tools and techniques presented in this article, the process of making experiments computationally reproducible should incur minimal overhead on their research. In particular, it is our hope that the solutions put forth in this article can help overcome all the current hurdles to seamless computational reproducibility. These solutions are discussed under three main themes in the following. Version control systems keep track of changes in the codebase as well as eliminate the issue of multiple concurrent versions of the experiments. Therefore, in order to support organized and understandable codebases, we first present the best tools for implementing version control. Next, in an effort to ensure that dependencies are accurately and easily shared, we discuss the simplest tools for dependency management that allow the researchers to disseminate their exact computational environment to others. Finally, in order to make the codebase easily explorable and provide thorough instructions on how to computationally reproduce the experiments, we offer some advice on the best ways to document and publish the code for sharing with the rest of the research community.

IV. Managing a Developing Experiment

Research is all about taking incremental steps towards a result. This is true when trying to prove theorems as well as in developing computational experiments. In practice the algorithms that get applied to datasets always need some level of tuning in order to run the computational experiments correctly or obtain the best possible results. Oftentimes one wants to investigate how changing a certain piece of the

code (e.g., a specific hyperparameter) alters the results without losing the current version of the code they have. A naïve solution to this problem is to make a copy of the source code with the desired changes without deleting the original. For example, a file named `my_algo.py` containing an implementation of the main algorithm for the experiment might get copied and renamed to `my_algo_1.py` with a few subtle changes made within it. Later the researchers may be interested in changing things a little differently but still keep `my_algo.py` and `my_algo_1.py` just in case the new version performs worse, so they create another file named `my_algo_2.py`, and this pattern repeats for multiple versions of the file. Finally, a few months after the work is published, someone might ask about how the figures were produced and the original researchers are left scrambling through up to a dozen different versions of the codebase, trying to find the one that actually produced the right results. Clearly this is not a good solution and can quickly become an unwieldy situation, forcing one to re-run multiple versions of the codebase while trying to find the correct one. Though this ad-hoc approach may have been feasible for small computational experiments that only take seconds or minutes to finish computing, with the rise in larger datasets and higher wall-clock time per computational experiment, re-running multiple versions of the experiments until one finds the right version can take days or even weeks. This is the precise problem we found in one of our codebases that we tried to computationally reproduce, which had multiple versions of the same experiment. Even the original researchers on the project were not able to recall the version that corresponded to figures in the published work. Given that some of these computational experiments took up to five days to finish running, finding the version that reproduced the original plots took us weeks.

Another common hurdle that researchers must overcome is trying to develop computational experiments with collaborators. Due to the nature of computational research, developing computational experiments collaboratively can be done without having to physically share the same computational resources, lab or even the same country. While this certainly makes collaboration easier compared to other scientific domains, collaborating in the development of a codebase is still not a simple task. Some of the current solutions that researchers employ are faulty and have many drawbacks. For instance a researcher may treat source code like any other file and rely on tools such as Google Drive [26], DropBox [27], Microsoft OneDrive [28] and email to share their code amongst collaborators. While these tools are great for sharing images, documents and other forms of media, they are not the best tools for sharing a developing codebase, as they require the collaborators to constantly download and then upload the codebase every time a change has been made. There is also a responsibility on the one making the changes to alert all other collaborators in order for them to re-sync their codebases. In short, this mode of collaboration can

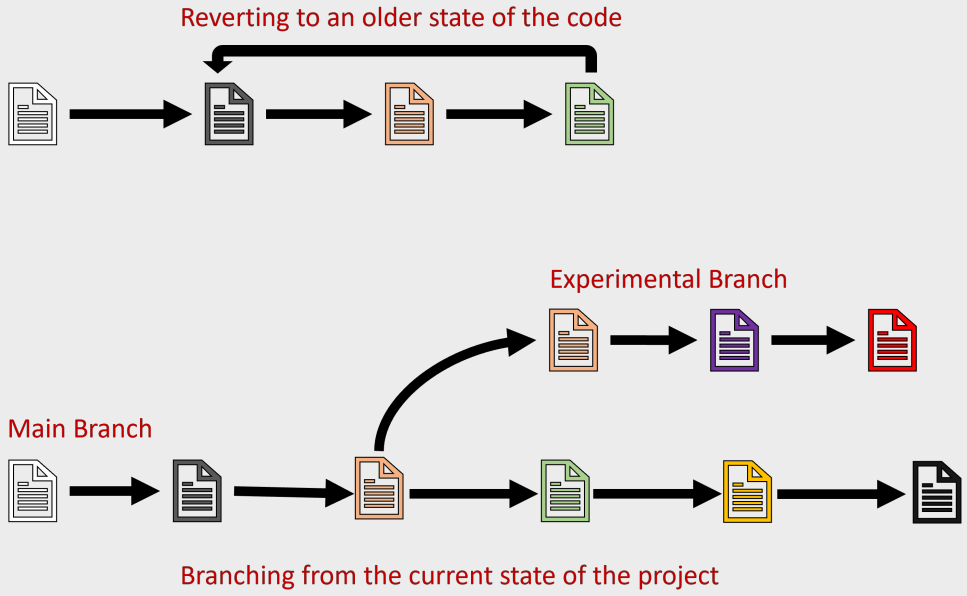
actually slow down collaboration and is prone to errors because it makes it very cumbersome to track changes made in the codebase and ensure that everyone is using the latest version of the codebase.

One of the best solutions to both of the aforementioned problems is an effective use of a good version control system, the most popular being `git`, a free and open-source distributed version control system known for its efficiency compared to other version control systems, making it easy to keep track of all the changes being made to an experiment throughout the development process. Due to its popularity and utility `git` integrates well with many online code hosting services, such as GitHub [29], BitBucket [30], and GitLab [31]. The advantage of `git` is that it allows one to track every change that has been made to any file in the codebase, often referred to as the *repository*. Once one of the files in the repository being tracked has been changed, the researcher can then move that file or set of files to the *staging* area. From the staging area the changes are then *committed*, which assigns the current state of the repository a unique SHA-1 hash and saves it in a history database so that it can be compared to future versions of the codebase or recovered when things go wrong. These commits also contain information of who made the changes and are accompanied by a short comment or message to explain what change were made and why. By committing regularly, a researcher is able to traverse through all changes of the codebase and see precisely which lines of code were changed, by whom they were changed, and a short description of why they were changed. The `git` branching feature also lends itself very well to experimenting and trying out different parameters or techniques. By *branching*, one can create an alternate tracking history of the repository starting from the current commit, and from there one can edit the code and run the experiments without affecting the current implementation on the main branch.

However, `git` truly shines in a collaborative setting. This is primarily due to its design as a distributed version control system, which means that it provides each collaborator a full commit history of the entire repository. When used jointly with a code hosting platform, such as GitHub [29], BitBucket [30], or GitLab [31], it allows researchers to upload their repository to the internet and *pull* down changes from the hosting service to their local environment every time they intend to develop on the codebase. This ability to pull down the changes is substantially better than the traditional approach of re-downloading a codebase from cloud storage providers because it eliminates the need to ask collaborators about what was altered or alert other collaborators of the changes implemented. This is because every commit that was made is already documented by the commit message accompanying it, along with the name of the person who made the commit. Furthermore, due to the efficient design of `git`, pulling down changes from a repository only takes seconds even if numerous changes were made; this is in contrast to re-downloading an entire codebase from a cloud storage provider, which can take minutes every time. Additionally git is

intelligently designed to handle merging the changes between the current version of the code on one's local machine and the updated codebase online, making the process very seamless and facilitating efficient asynchronous collaboration.

While some researchers may argue that `git` is a complex tool which ultimately impedes their productivity, this is only temporary and we believe that that it is worth the investment for computational researchers. Furthermore, most researchers in machine learning and signal processing collaborate extensively with industry where version control is expected to be used in maintaining any codebase. Perhaps the most appealing feature of using `git` for version control over other alternatives is its widespread use among programmers. Over the years, this has resulted in numerous resources all over the internet in the form of videos, blog posts [32], and books [33]. Additionally, online communities such as StackOverflow [34] provide answers to nearly any confusion one may have about `git` and its features. There are also a number of graphical interface tools for using `git` for those not yet comfortable using the command line. While `git` is an extremely flexible tool and can be a tremendous aid in writing organized and manageable code for experiments across multiple researchers working together, it is only effective when utilized properly. Researchers must ensure that they are committing changes frequently and are providing providing concise and accurate descriptions of the changes made for easy reference as the codebase is developed.



Version Control: By effectively using version control and making frequent commits, researchers do not have to save multiple versions of the same file in their codebase, because they are always able

to easily revert back to an earlier commit. With version control, specifically `git`, one is also free to pursue multiple versions of the experiment without impinging on each other. This is most effectively done by branching, which allows one to create multiple version histories.

Going beyond version control, the next important step for ensuring reproducibility is to make sure that those trying to reproduce the code are able to capture the same exact computational environment as the one used by the original researchers. This is the topic of our next discussion.

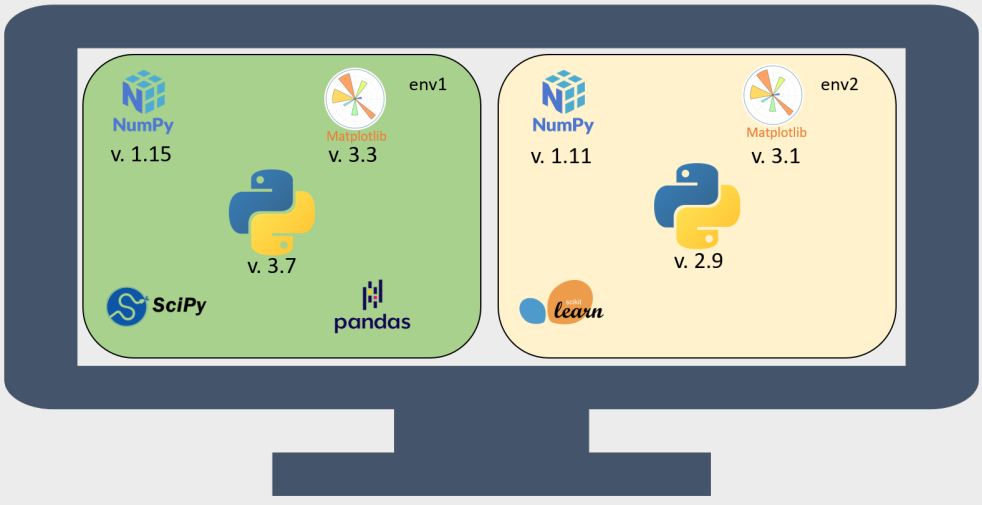
V. Managing Dependencies

One of the simplest ways to ensure a robust and reproducible codebase is to try and minimize the use of external packages and libraries in the code. However, due to the immense utility of modern software libraries, the codebases for modern signal processing and machine learning experiments are seldom developed without the use of several external libraries. These include popular scientific programming and machine learning libraries such as NumPy [10], Scikit-learn [11], TensorFlow [13], CVX [35], and Tensor Toolbox [36]. Although there are multiple programming languages available for scientific computing, such as R [37], Julia [38], and MATLAB [12], we focus the discussion specifically on the Python programming language as it is free, open source and widely popular in the machine learning community. However, the techniques discussed in this section can still be utilized for other popular programming languages.

Without first knowing which exact dependencies need to be installed on a researcher's machine, it becomes impossible to reproduce results. The reproducer must not only be aware of what dependencies are being used, but also the precise version being used at the time when the results were originally generated. One popular way to encapsulate the original researcher's computational environment perfectly is by using Docker [39], as suggested in [40]. While Docker is a powerful containerization tool and it does indeed solve the problem of dependency management by encapsulating the original researcher's computational environment into isolated *containers*, it could be too much of an overhead for researchers who have no prior experience using Docker or managing large software projects. This is especially true in labs and research groups where researchers focus primarily on the mathematical and algorithmic aspects of their research and only utilize a small codebase for experiments, which relies on only a few dependencies. In these computational experiments one typically does not attempt to synthesize and manage multiple programming languages, frameworks and dependencies, as is often the case in real-world commercial applications, making Docker-based solutions an overkill.

Therefore, a more appropriate tool would be a simple and light dependency manager similar to `pip`, the default dependency manager for Python. But the most fitting environment and dependency manager for computational experiments is `conda` [41], an open-source environment and package management tool.

Although `conda` can be used independently in any codebase, it is automatically included in the Anaconda [42] distribution of Python, which is already the distribution of choice for many computational researchers. Unlike other package managers, `conda` is specifically designed to easily manage the dependencies most commonly encountered in scientific computing, and overcomes the many shortcomings of `pip`. One major advantage of using `conda` is that when one attempts to installs a new package, `conda` ensures that all the requirements for this new package are met before adding it, and if this is not the case an error is shown immediately with steps on how to rectify the issue. This is contrary to `pip`, which does not check this condition before installing a new package and can result in unexpected errors later during development. However, `conda`'s greatest advantage is allowing one to create independent *virtual environments* for each experiment, so that all projects do not share the same global dependencies. This in turn ensures that each environment contains only the packages that are absolutely needed for the current experiment associated with it and nothing more. Creating these virtual environments is also important because when the version of a package from one project gets upgraded on the researcher's machine, it will not interfere with the current version of that same package in the other environments, allowing the original computational environment on which the experiment was carried out to be preserved. Finally, while Python's standard library does include its own virtual environment manager through the `venv` module, `conda` is unique in that it allows one to create environments with different versions of Python itself.



Conda Environments: A conceptual diagram that illustrates the usefulness of conda environments for reproducible research. On the left, "env1" is a python environment with four different libraries commonly used in machine learning experiments. On the right, the environment "env2" has some

environment.yml

```yaml
name: experiment1
channels:
  - defaults
dependencies:
  - python=3.7.7
  - tensorflow=1.15
  - numpy
  - jupyterlab
  - matplotlib
```

Fig. 1. An example of an `environment.yml` file with common libraries and dependencies used in typical machine learning experiments.

of the same libraries but with different version numbers and a different version of python itself; this could, for example, be an environment for an older variant of the experiment. Both these environments are independent of each other and preserve the computational environment that was originally used to produce experimental results. They can also be easily exported and shared with outside researchers through an `environment.yml` file. The environment can be reconstructed on a new computer using the command ``conda env -f environment.yml''.

Based on the above, we recommend using `conda` for managing dependencies in reproducible experiments. This involves creating an `environment.yml` (or a similarly named) file for each project, which specifies the necessary dependencies, and then constructing a new environment based on the specifications in this file, which can be done with a simple command. This should be done before any code for the experiment is written in order to ensure no dependencies are unaccounted for. If one wants to later add a dependency they can do so by adding it to their `environment.yml` file and updating their environment according to the new additions made to the file. Then, when another independent researcher wishes to

run the experiment, they can quickly and easily reconstruct the same `conda` environment with all the necessary dependencies by simply using the `environment.yml` file. Additionally, one can specify the exact version number for the important dependencies in the "environment.yml" file to ensure that when the "conda" environment gets reconstructed on another machine (potentially with another operating system) the same dependency version is used. Also, for those who prefer some other dependency manager as opposed to `conda`, they can still inspect the plain text yaml file to see all the necessary dependencies with their respective versions. A typical "environment.yml" for a machine learning experiment is shown in Figure 1, demonstrating its readability and effectiveness in organizing dependencies. Here we have *pinned* the version for TensorFlow to ensure that this same version is always preserved when the environment is reconstructed on a different machine. By utilizing this tool, managing dependencies becomes very straightforward and makes it possible for others to replicate the same computational environment that one had used when originally running the experiment, without the need to resort to trial and error or to contact the original researchers.

Proper version control and dependency management tools are critical first steps to promoting reproducibility throughout development of an experiment. However, ensuring reproducibility for the foreseeable future is made most probable through properly shared code and data, detailed documentation and thoughtful organization of the codebase developed. We now shift our focus to this aspect of reproducibility.

VI. Sharing Code

While version control and dependency management make it possible for others to run the code and help to keep the codebase organized, this does not provide any information about the order in which the scripts should be executed and what computational resources were utilized in obtaining the original results. Furthermore, when researchers look to reproduce other's experiments, they do not necessarily want to reproduce each and every figure in the paper. This is especially true for papers with multiple experiments. Sometimes they are only interested in a specific plot or the implementation of a particular algorithm that was described in the original work. Therefore it is important to ensure that one's experiments do not get crammed into a single source file that includes preprocessing, algorithmic implementations, analysis and visualization. Indeed, by making the codebase modular and organizing the project structure carefully, as discussed in the following, it becomes significantly easier for others to download and inspect what they want to reproduce from the codebase without too much digging.

Project Structure: Typically, a computational experiment in signal processing and machine learning can be broken down into three parts:

1) Preprocessing of data or generation of synthetic data.

2) Execution of the actual experiment on the data.
3) Analysis of results and generation of figures.

It is often advantageous to keep the project folder organized in a similar fashion by creating separate scripts, or even subfolders, for each of these tasks. Keeping the codebase modular and structured in this manner allows others looking to reproduce a particular set of results to do so without needing to analyze or execute the entire codebase. For example, in our lab's experiments on learning mixtures of separable dictionaries for tensor data [43], we found it necessary to split the experiments up into different subfolders as the original work [44] compared four different algorithms on both synthetic and real datasets. Object-oriented design within the code can also be helpful as this increases the modularity of the experiments and allows extensions to be developed by other researchers naturally. Last, but not the least, the data used in the experiments must also be shared correctly; this involves ensuring that not only the raw data get shared but also all the intermediate steps or scripts used to preprocess the data before using them in the experiments are shared.

Coding Standards: Additionally, it is worth following a set of guidelines for clear and concise comments that can describe every function or class definition in the codebase. For Python the convention is to add a *docstring* [45] to each module, as many text editors and integrated development environments (IDEs) often search for these in the project files to allow programmers to quickly inspect what a function is doing without actually going to its destination in the source code. This should be accompanied by standard coding practices, such as proper variable naming conventions and indentations, which can make the codebase easier to read and interpret by others.

Documentation: The most important thing that must accompany the project being shared is a detailed README file. We recommend that the README contain all the following elements:

1) Brief description of the project and related paper(s).
2) Steps to install necessary dependencies, e.g., via `conda` and `environment.yml` file.
3) Computational resources used to run the experiments along with their respective runtimes.
4) Scripts used to preprocess the data, in the case of real data, or generate them for the case of synthetic data.
5) Description of which scripts one should run for which experiments and how to then plot the results.

We found that each of these pieces of information were necessary when attempting to reproduce the results from a codebase. Such documentation that accompanies the code provides a full picture of how an experiment should be run and how one actually acquires the desired results without having to spend much time laboring through every detail in the code.

Publicization: The way in which the code is publicized also needs careful consideration to ensure ease of access and, most importantly, permanency. Typically experiments are shared by hosting the accompanying source code on the lab's website or on a code hosting platform, such as GitHub, GitLab or BitBucket. However, posting it on the lab's website is unlikely to be the most robust solution; for instance, if a researcher leaves his or her position, their website often disappears along with the code. Code hosting services, however, are linked to an account and repositories can be easily transferred to different owners with minimal hassle. Sharing the code this way also allows others to fork the repository and develop further on it. Additionally, others can add their comments or questions about the code in the form of "Issues." These discussions become public for anyone else viewing the repository as well, which can eliminate redundant questions that the community may have. One of the biggest issues with both of these solutions is that they are still dependent on the organization hosting the website. This means that if the hosting website were to ever disappear, the code hosted on it would go with it. But permanency on the internet is necessary for ensuring that the results are reproducible for the forseeable future. One way to ensure permanency is to assign the codebase a digital object identifier (DOI) to give it a permanent presence on the internet, which can be done with tools such as Zenodo [46].

VII. Conclusion

While there has been much discussion in the literature about the reproducibility crisis in computational research, not enough emphasis has been given on the best practices and techniques to solve this problem with established tools. The solutions to computational reproducibility have been especially unaddressed for the more theoretically bent research groups within the signal processing and machine learning community that typically develop smaller computational experiments compared to other computational sciences. In this article, we presented the main pitfalls to achieving reproducible experiments and then provided common tools and techniques that can be used to overcome each of those pitfalls, while bearing in mind that making experiments reproducible can entail extra effort that may divert our attention away from our primary task of research. By utilizing the right tools for version control and dependency management as well as careful structuring and documentation when sharing the codebase, we can work towards ensuring that every computational experiment in our research is readily reproducible.

Acknowledgements

The authors gratefully acknowledge the support of the NSF (CCF-1453073, CCF-1907658, CCF-1910110, OAC-1940074) and the ARO (W911NF-17-1-0546, W911NF-21-1-0301).

SHORT BIOGRAPHIES

Joseph Shenouda (jshenouda@wisc.edu) received his B.S. degree in Electrical & Computer Engineering in 2021 from Rutgers University–New Brunswick. He is currently a graduate student at the University of Wisconsin–Madison in the Electrical & Computer Engineering department. His research interests include graph signal processing and machine learning.

Waheed U. Bajwa (waheed.bajwa@rutgers.edu) has been with Rutgers University–New Brunswick, NJ since 2011, where he is currently an associate professor in the Departments of Electrical & Computer Engineering and Statistics. His research interests include statistical signal processing, high-dimensional statistics, and machine learning. He is a senior member of the IEEE.

REFERENCES

[1] V. Stodden, F. Leisch, and R. D. Peng, *Implementing Reproducible Research*. CRC Press, 2014.

[2] E. Raff, "A Step Toward Quantifying Independently Reproducible Machine Learning Research," in *Proc. Advances in Neural Information Processing Systems*, vol. 32, 2019, pp. 5485–5495.

[3] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[4] S. N. Goodman, D. Fanelli, and J. P. Ioannidis, "What does research reproducibility mean?" *Science translational medicine*, vol. 8, no. 341, pp. 341ps12–341ps12, 2016.

[5] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[6] V. C. Stodden, "Trust your science? open your data and code," 2011.

[7] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[8] J. Pineau *et al.*, "Improving reproducibility in machine learning research: a report from the neurips 2019 reproducibility program," *Journal of Machine Learning Research*, vol. 22, 2021.

[9] N. P. Rougier *et al.*, "Sustainable computational science: the rescience initiative," *PeerJ Computer Science*, vol. 3, p. e142, 2017.

[10] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[11] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[12] Mathworks, "MATLAB." [Online]. Available: www.mathworks.com

[13] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[14] H. Monajemi, D. L. Donoho, and V. Stodden, "Making massive computational experiments painless," in *Proc. IEEE International Conference on Big Data*, 2016, pp. 2368–2373.

[15] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature News*, vol. 533, no. 7604, p. 452, May 2016, section: News Feature. [Online]. Available: http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970

[16] P. Vandewalle, J. Kovacevic, and M. Vetterli, "Reproducible research in signal processing," *IEEE Signal Processing Magazine*, vol. 26, no. 3, pp. 37–47, May 2009.

[17] E. Bjornson, "Reproducible research: Best practices and potential misuse [Perspectives]," *IEEE Signal Processing Magazine*, vol. 36, no. 3, pp. 106–123, May 2019.

[18] F. Markowetz, "Five selfish reasons to work reproducibly," *Genome Biology*, vol. 16, no. 1, p. 274, Dec. 2015.

[19] H. A. Piwowar, R. S. Day, and D. B. Fridsma, "Sharing detailed research data is associated with increased citation rate," *PLoS ONE*, vol. 2, no. 3, p. e308, Mar. 2007.

[20] Y. AlNoamany and J. A. Borghi, "Towards computational reproducibility: researcher perspectives on the use and sharing of software," *PeerJ Computer Science*, vol. 4, p. e163, 2018.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[22] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, pp. 400–407, 1951.

[23] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks, Tricks of the Trade, Reloaded*, 2nd ed. Springer, January 2012, vol. 7700, pp. 430–445.

[24] A. Brinckman *et al.*, "Computing environments for reproducibility: Capturing the "Whole Tale"," *Future Generation Computer Systems*, vol. 94, pp. 854–867, May 2019.

[25] M. Konkol, D. Nüst, and L. Goulier, "Publishing computational research - a review of infrastructures for reproducible and transparent scholarly communication," *Research Integrity and Peer Review*, vol. 5, no. 1, p. 10, Dec. 2020.

[26] Google LLC, "Google Drive Website." [Online]. Available: https://drive.google.com

[27] Dropbox Inc., "Dropbox Website." [Online]. Available: https://www.dropbox.com

[28] Microsoft Corp., "Microsoft OneDrive Website." [Online]. Available: https://onedrive.com

[29] GitHub, Inc., "GitHub Website." [Online]. Available: https://www.github.com

[30] Atlassian Corp., "Bitbucket Website." [Online]. Available: https://bitbucket.org

[31] GitLab Inc., "GitLab Website." [Online]. Available: https://gitlab.com

[32] C. Duan, "Understanding Git Conceptually." [Online]. Available: https://www.sbf5.com/~cduan/technical/git/

[33] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Apress, Nov. 2014.

[34] Stack Exchange Inc., "Stack Overflow Website." [Online]. Available: https://stackoverflow.com/

[35] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 2.1," Mar. 2014. [Online]. Available: http://cvxr.com/cvx

[36] B. Bader, T. G. Kolda, et al., "Tensor toolbox for MATLAB, version 3.1," Jun. 2019. [Online]. Available: www.tensortoolbox.org

[37] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2020. [Online]. Available: https://www.R-project.org/

[38] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671

[39] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2:2, Mar. 2014.

[40] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[41] Anaconda Inc., "Conda," 2021, vers. 4.10.1. [Online]. Available: https://anaconda.org/anaconda/conda

[42] Anaconda, Inc., "Anaconda software distribution," 2020, vers. 2-2.4.0. [Online]. Available: https://docs.anaconda.com/

[43] J. Shenouda, M. Ghassemi, Z. Shakeri, A. D. Sarwate, and W. U. Bajwa, "Codebase—Learning mixtures of separable dictionaries for tensor data: Analysis and algorithms," GitHub Repository, 6 2020. [Online]. Available: https://github.com/INSPIRE-Lab-US/LSR-dictionary-learning

[44] M. Ghassemi, Z. Shakeri, A. D. Sarwate, and W. U. Bajwa, "Learning mixtures of separable dictionaries for tensor data: Analysis and algorithms," *IEEE Transactions on Signal Processing*, vol. 68, pp. 33–48, 2020.

[45] python-dev, "PEP 257 – Docstring Conventions." [Online]. Available: https://www.python.org/dev/peps/pep-0257/

[46] CERN, "Zenodo Website." [Online]. Available: https://zenodo.org/