The Journal of Open Source Software

Deckard: A Declarative Tool for Machine Learning Robustness Evaluations

Charles Meyers[1*]

1 UmeåUniversity Department of Computer Science, UmeåUniversity * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Deckard is a modular software toolkit designed to streamline and standardize experimentation in adversarial machine learning. It provides a flexible, extensible framework for defining, executing, and analyzing end-to-end machine learning pipelines with a particular focus on adversarial robustness. Built on top of the Hydra configuration system, Deckard supports declarative YAML-based configuration of data preprocessing, model training, and adversarial attack pipelines, enabling reproducible, framework-agnostic experimentation across diverse machine learning settings.

In addition to configuration management, Deckard includes a suite of utilities for distributed and parallel execution, automated hyperparameter optimisation, visualisation, and result aggregation. The tooling abstracts away much of the engineering overhead typically involved in adversarial ML research, allowing researchers to focus on algorithmic insights rather than implementation details. Deckard also facilitates rigorous benchmarking by maintaining an auditable trace of configurations, random seeds, and intermediate outputs throughout the experimental lifecycle.

The system is compatible with a variety of ML frameworks and adversarial attack libraries, making it a suitable backend for both large-scale automated testing and fine-grained empirical analysis. By providing a unified interface for experimental control, Deckard accelerates the development and evaluation of robust models, and helps close the gap between research prototypes and verifiable, reproducible results

Statement of need

While tools such as MLflow, Weights & Biases, Optuna, Kubernetes provide essential infrastructure for model tracking and experiment management, Deckard occupies a different position in the machine learning ecosystem——focusing specifically on configurable, adversarially robust experimentation.

Unlike MLflow and Weights & Biases, which emphasize logging, visualization, and reproducibility for various ML frameworks, deckard enforces reproducibility by construction through its declarative, YAML-driven configuration system built on Facebook's hydra configuration management tool. In contrast to cloud-management software like Kubernetes—which is a general-purpose container orchestration platform—Deckard abstracts away orchestration details and offers native support for parallel and distributed experimentation, tailored to ML workflows involving attack/defense cycles, model retraining, or optimisation. While deckard integrates tightly with IBM's Adversrial Robustness Toolbox (`art`), the software is designed to be easily extensible to other attack frameworks. The human- and machine-readable parameter configuration system allows researchers to declaratively define end-to-end pipelines that span data sampling, preprocessing, model training, attack generation, defense evaluation, multi-objective optimisation, and visualisation. Tools like Ray, Optuna, or Sacred offer components of this pipeline (e.g., hyperparameter search or configuration management), but lack unified support for adversarial ML, verification, or auditability at scale. Deckard complements these existing tools, and in many cases can be integrated alongside them, but its primary contribution is in automating and verifying adversarial machine learning experiments in a way that is both extensible and framework-agnostic.

Usage

Various versions of this software have been used in several published and unpublished works by the author of this paper. The first published work, now reproducible via the `examples/tf2' folder, includes a large survey of attacks and defences against canonical datasets and models[@meyers2023safety]. The second work analysed the run-time requirements of attacks against a particular model before after retraining against those attacks[@meyers2024massively] (reproducible via`examples/security'). The third paper formalised a method for estimating the time-to-failure of a given model against a suite of attacks and introduce a metric that quantifies the ratio of attack and training cost(Meyers et al., 2023) (reproducible via `examples/pytorch'). Furthemore, an unpublished work uses this time-to-failure model as a mechanism for analysing the cost efficacy of various hardware choices in the context of adversarial attacks (reproducible via`examples/power')(C. Meyers et al., 2024). A fifth and final work exploits the tooling to train a custom model that is designed to run client-side by using compression algorithms to measure the distance between text (reproducible via 'examples/gzip'.

Experiment Management

Typically machine learning pipelines are composed of long and complex pipelines that are highly dependent on a number of parameters that must be configured by either the model builder or attacker. Due to the difficulty of optimising popular models (*i.e.* neural networks), it is often necessary to tune a model using hundreds or thousands of indivudal configurations. In addition, even simple models are often part of various long and complex data and software pipelines. Generally, one of many benchmark datasets is first sampled, then preprocessed, sent to a model, with optional pre- and post-processing defences, and then scored according to some chosen metric which may include the performance against any number of adversarial attacks. Each stage in this example pipeline might include 10s or 100s of possible configurations that must be exhaustively tested. As such, this problem scales drastically as we include more and more stages in a pipeline since each configuration must be compared against each other. Not only does `deckard` provide a standard way to document and configure these parameters, it gives each experiment an auditable identifier that is difficult to forge.

Reproducibility and Auditability

The software package presented here provides a machine- and human-readable format for creating reproducible and auditable experiments, as required by various regulatory and legal frameworksLegislature of the United States (1998). In addition, several examples connected to both published and unpublished work live in the `examples` folder in the repository, allowing for easy reproducibility of several extensive sets of experiments across several popular machine learning software frameworks. The `power` example provides a reproducible way to run a suite of adversarial tests using popular cloud-based platforms and the `pytorch` and `security` examples provide examples of both CPU and GPU-based parallelisation, respectively.

The parameters file for each experiment ensures that a given pipeline can be reproduced and the standardised format allows us to derive an hash value that is hard to forge but easy to verify. Not only does this hash serve as an identifier to track the state of an experiment, but also serves as a way to audit the parameters file for tampering. Likewise, by using `dvc` to track any input or output files specified in the parameters file, the software associates each score file with a identifier that is easy to track and verify and hard to forge, ensuring that forged or modified results are easy to spot in version-controlled experiment repository.

Parallel and Distributed Design

Since machine learning projects can exploit specialized hardware such as multi-core processors or GPUs, and often rely on clusters of machines for large-scale data processing, it was necessary to enable parallel and distributed experiment execution and model optimization. By leveraging the `hydra` configuration framework, `deckard` automatically supports optimization libraries like `nevergrad`, `ax`, and `optuna`, making the software modular and extensible. Additionally, experiments can be managed using a variety of popular job schedulers, including `joblib`, `Ray`, `RQ`, and `slurm`.

By using a declarative design, a given set of experiments can be specified once and executed seamlessly across different backends without modification to the underlying codebase. This makes `deckard` both adaptable and scalable, suitable for use on personal laptops, multi-GPU servers, or large-scale HPC clusters. When configured appropriately, experiment batches can be parallelized using `joblib` or scheduled as distributed tasks using `Ray`, `RQ`, or `slurm`, enabling massive parameter sweeps, ensemble evaluations, or adversarial robustness tests to be executed in parallel—reducing turnaround time while maintaining strong guarantees on reproducibility and auditability. The design of the presented software prioritizes clarity and maintainability by capturing each experimental configuration as a YAML artifact, making both successful and failed runs equally traceable and shareable. This approach transforms experiment tracking from an afterthought into a first-class component of the trustworthy machine learning research workflow.

Citations

Citations to entries in paper.bib should be in [rMarkdown](http://rmarkdown.rstudio.com/authoring_bibliographies_and_citations.html) format.

If you want to cite a software repository URL (e.g. something on GitHub without a preferred citation) then you can do it with the example BibTeX entry below for (**?**).

For a quick reference, the following citation commands can be used: - `@author:2001` -> "Author et al. (2001)" - `[@author:2001]` -> "(Author et al., 2001)" - `[@author1:2001; @author2:2001]` -> "(Author1 et al., 2001; Author2 et al., 2002)"

Figures

Figures can be included like this: Caption for example figure. and referenced from text using [section](#figures) .

Figure sizes can be customized by adding an optional second parameter: Caption for example figure.

Funding

Financial support has been provided in part by the Knut and Alice Wallenberg Foundation grant number 2019.0352 and by the eSSENCE Programme under the Swedish Government's Strategic Research Initiative.

The Journal of Open Source Software

Acknowledgements

The author would like to thank Aaron MacSween, Abel Souza, and Mohammad Saledghpour Reza for their guidance in software design principles. In particular, the author appreciates Mohammad's code and documentation regarding cloud-based and other Kubernetes deployments.

References

Legislature of the United States. (1998). *Children's online privacy protection act.*

Meyers, C., Sedghpour, M. R. S., Löfstedt, T., & Elmroth, E. (2024). *A training rate and survival heuristic for inference and robustness evaluation (TRASHFIRE)*. https://arxiv.org/abs/2401.13751

Meyers, Reza, Löfstedt, & Elmroth. (2023). A systematic approach to robustness modelling. *Springer Artificial Intelligence Review.*