REPRODUCIBLE EXPERIMENTS

THE “WHY” , THE “WHAT” AND SOME “HOWs”.

A MultiX Thematic Session Talk by Ujjwal Sharma

THE WHY

Is AI leading to a reproducibility crisis in science?

Scientists worry that ill-informed use of artificial intelligence is driving a deluge of unreliable or useless research.

By Philip Ball

Association for Computing Machinery

ACM EIG on Reproducibility and Replicability

ABOUT **WORKING GROUPS** **TAKING THE PULSE**

Reproducibility and Replicability

ACM Emerging Interest Group

Fostering a diverse and inclusive community around the issues of reproducibility and replicability of computational research.

Home > Home

NEWSLETTER

Reproducibility Retro! is a resource on all things reproducibility. It is written for the ACM community and for any stakeholder seeking to increase transparency, reproducibility, and reusability of research.

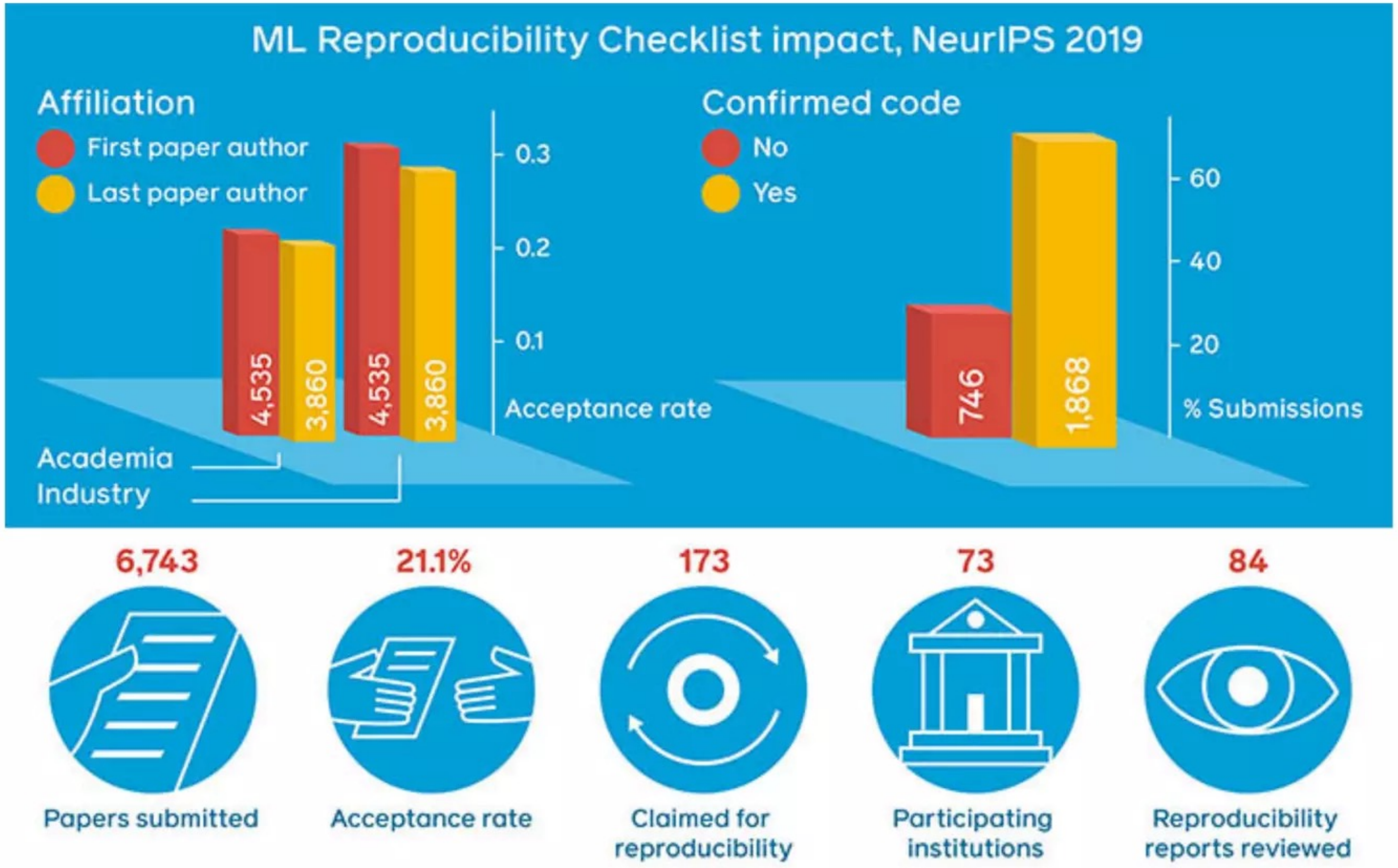
Access on ACM DL

ACTIVITIES

The EIG proposes two working groups, one to focus on conference planning and execution activities, and the other to coordinate discussions and efforts regarding reproducibility (best/better) practices.

Join the EIG

REPRODUCIBLE RESEARCH



NeurIPs 2019 Reproducibility Checklist

REPRODUCIBILITY CHALLENGE

- Difficult to reproduce the results of a paper.
- Missing data, Model weights, Scripts, etc.
- Hyperparameters, Features, Data, Vocabulary and other artifacts are lost.
- Impossible to recreate the *secret sauce*.

TRADITIONAL SOFTWARE VS. MACHINE LEARNING

- Continuous, Iterative process, Optimize for metric
- Quality depends on data and tuning parameters
- Experiment tracking is difficult
- Over time data changes, model drift
- Compare + combine many libraries and models
- Diverse deployment environments

WHAT IS REPRODUCIBILITY

SOFTWARE REPRODUCIBILITY

1. In the context of software:

"Reproducibility is the capacity for an individual to reproduce a computational experiment conducted by another party. This involves employing identical software and datasets to replicate the experiment, while retaining the flexibility to modify certain components—namely, the software and/or the data—facilitating a deeper comprehension of the experiment and its constraints."

MULTI-LAYER SOFTWARE STACKS

- Almost all software stacks used in computational science have a nearly universal multi-layer structure:
  1. Project-specific software: whatever it takes to do a computation using software building blocks from the lower three levels: scripts, workflows, computational notebooks, small special-purpose libraries and utilities
  2. Discipline-specific research software: tools and libraries that implement models and methods which are developed and used by research communities
  3. Scientific infrastructure: libraries and utilities used for research in many different disciplines, such as LAPACK, NumPy, or Gnuplot
  4. Non-scientific infrastructure: operating systems, compilers, and support code for I/O, user interfaces, etc.

MULTI-LAYER SOFTWARE STACKS

- Almost all software stacks used in computational science have a nearly universal multi-layer structure:
  1. Project-specific software: whatever it takes to do a computation using software building blocks from the lower three levels: scripts, workflows, computational notebooks, small special-purpose libraries and utilities
  2. Discipline-specific research software: tools and libraries that implement models and methods which are developed and used by research communities
  3. Scientific infrastructure: libraries and utilities used for research in many different disciplines, such as LAPACK, NumPy, or Gnuplot
  4. Non-scientific infrastructure: operating systems, compilers, and support code for I/O, user interfaces, etc.

Just addressing project-specific software (the top layer) isn't enough to solve software collapse; the lower layers are still likely to change.

WHAT CAN YOU DO?

- Prepare for Failure : Regard your code and its associated dependencies susceptible to failure at any moment. In the event of a failure, be prepared to start from scratch.
- Repair – whenever foundations start to move under your project (library updates, foundational changes, etc.), duly perform the required repairs.
- Adaptability: Design your project to withstand disturbances, ensuring resilience to unforeseen challenges. Don't rely excessively on idiosyncratic attributes of your development environment.
- Stability: Opt for secure and dependable foundational elements for your foundations – OS, Compilers, etc.

10 Simple Rules for Reproducible Computational Research

1. For Every Result, Keep Track of How It Was Produced
2. Avoid Manual Data Manipulation Steps
3. Archive the Exact Versions of All External Programs Used
4. Version Control All Custom Scripts
5. Record All Intermediate Results, When Possible in Standardized Formats
6. For Analyses That Include Randomness, Note Underlying Random Seeds
7. Always Store Raw Data behind Plots
8. Generate Hierarchical Analysis Output, Allowing Layers of Increasing Detail to Be Inspected
9. Connect Textual Statements to Underlying Results
10. Provide Public Access to Scripts, Runs, and Results

Record Everything

Automate Everything

Sandve GK, Nekrutenko A, Taylor J, Hovig E (2013) Ten Simple Rules for Reproducible Computational Research. PLoS Comput Biol 9(10): e1003285. doi:10.1371/journal.pcbi.1003285

CORE IDEAS

- Version Control for *EVERYTHING*: Development Environment, Experiments and Code (and even Data).
- Ensure a replicable, deterministic path from data + code to results.
- Experiment logging:
  - Log Evaluation measures.
  - Freeze source of possible variation via fixed random seeds.
  - Record intermediate results.

WHAT CAN YOU CONCRETELY DO?

VERSION CONTROL YOUR CODE

- Use a Version Control System (VCS) such as Git or Mercurial to monitor all your experimental code. Ensure that all relevant code files are tracked and committed before initiating each experiment.
- Each attempt at implementing a new architecture, adjusting hyperparameters, etc., should be associated with a unique repository state.
- Ensure that your Git repository is not in a 'dirty' state prior to code execution, as the reasons for avoiding this will soon become clear."

VERSION CONTROL YOUR ENVIRONMENT

- Many deep learning (DL) projects in Python employ a common mix of libraries and discipline-specific research software, such as PyTorch and NumPy, to achieve results.
- The conventional method involves tracking a requirements.txt file, which, when used in tandem with venv, allows the creation of a new environment.
- However, it's worth noting that 'pip' isn't an ideal dependency manager. There are more effective alternatives available, and it's recommended to explore them for a more robust solution.

Building identical conda environments

You can use explicit specification files to build an identical conda environment on the same operating system platform, either on the same machine or on a different machine.

Use the terminal for the following steps:

1. Run `conda list --explicit` to produce a spec list such as:

   ```
   # This file may be used to create an environment using:
   # $ conda create --name <env> --file <this file>
   # platform: osx-64
   @EXPLICIT
   https://repo.anaconda.com/pkgs/free/osx-64/mkl-11.3.3-0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/numpy-1.11.1-py35_0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/openssl-1.0.2h-1.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/pip-8.1.2-py35_0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/python-3.5.2-0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/readline-6.2-2.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/setuptools-25.1.6-py35_0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/sqlite-3.13.0-0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/tk-8.5.18-0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/wheel-0.29.0-py35_0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/xz-5.2.2-0.tar.bz2
   https://repo.anaconda.com/pkgs/free/osx-64/zlib-1.2.8-3.tar.bz2
   ```

2. To create this spec list as a file in the current working directory, run:

   ```
   conda list --explicit > spec-file.txt
   ```

↑ Back to top

POETRY

1. **Overview:**
    1. Poetry is a Python packaging and dependency management tool.
    2. Aimed at improving upon limitations of traditional pip.
2. **Dependency Management and Locking:**
    1. Declarative management using pyproject.toml.
    2. Generates a poetry.lock file for consistent builds.
3. **Single-File Configuration:**
    1. Simplifies project configuration with a single pyproject.toml file.
    2. Contrast with pip, which may use multiple files.

POETRY

- **Consistent Dependency Resolution:**
  - Uses a dedicated resolver for predictable environments.
  - Ensures consistent dependency resolution.
- **Semantic Versioning:**
  - Encourages the use of semantic versioning.
  - Enhances understanding of package compatibility.

POETRY

- **pyproject.toml:**
  - Project Configuration: Holds project metadata and configuration.
  - Dependency Declaration: Specifies dependencies and constraints.
  - Build and Tool Settings: Defines build and tool configurations.

- **poetry.lock:**
  - Dependency Locking: Locks down exact versions of dependencies.
  - Ensures Reproducibility: Guarantees consistent dependencies across environments.
  - Dependency Hashes: Includes hashes for security and integrity verification.

WHY?

- It helps you reliably build deterministic environments everywhere (local machine, snellius and even your supervisor's machine!)
- Commit pyproject.toml and poetry.lock to VCS to allow users to build an identical build environment.

VERSION CONTROL: EXPERIMENTS

- Versioning entire experiment requires versioning *ALL* of the artifacts on which it depends.
  - Code
  - Data
  - Runtime attributes (hyperparameters specified via the command line, GPU configuration, etc.)

(PYTORCH) LIGHTNING

1. Offers a standardized training loop, separating phases for clarity in code organization.
2. Automatic GPU scaling and support for multi-GPU training without manual intervention.
3. Callback system for easy customization of training behavior (e.g., logging, model checkpointing).
4. Native support for Automatic Mixed Precision, balancing speed and accuracy.
5. Seamless integration with tools like TensorBoard for easy experiment tracking.
6. Default configurations enhance experiment reproducibility.

Re-organize your code to move specific parts of the code into specialized functions

```python
import os
from torch import optim, nn, utils, Tensor
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
import lightning as L

# define any number of nn.Modules (or use your current ones)
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))


# define the LightningModule
class LitAutoEncoder(L.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        # it is independent of forward
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = nn.functional.mse_loss(x_hat, x)
        # Logging to TensorBoard (if installed) by default
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=1e-3)
        return optimizer


# init the autoencoder
autoencoder = LitAutoEncoder(encoder, decoder)
```

The Trainer can then invoke the right functions in the right order to train your model.

```
# setup data
dataset = MNIST(os.getcwd(), download=True, transform=ToTensor())
train_loader = utils.data.DataLoader(dataset)

# train the model.
trainer = L.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

You can do a lot now!

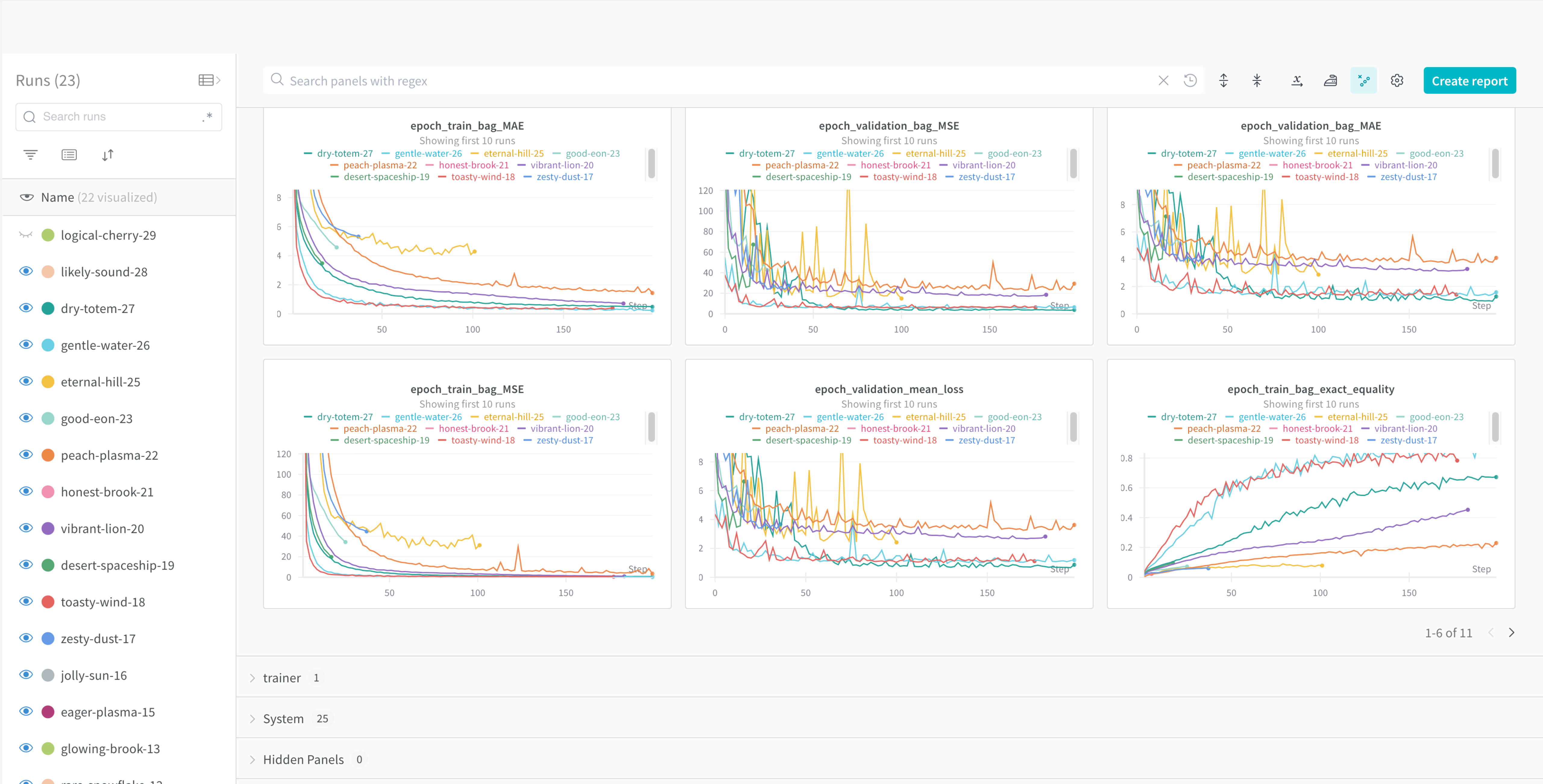
```
# train on multiple GPUs
trainer = L.Trainer(
    devices=4,
    accelerator="gpu",
 )

# train 1TB+ parameter models with Deepspeed/fsdp
trainer = L.Trainer(
    devices=4,
    accelerator="gpu",
    strategy="deepspeed_stage_2",
    precision=16
 )

# Rapid idea iteration
trainer = L.Trainer(
    max_epochs=10,
    min_epochs=5,
    overfit_batches=1
    fast_dev_run=1
 )

trainer = L.Trainer(callbacks=[StochasticWeightAveraging(...)])
```

All runs can be trivially logged.



logical-cherry-29

Description	What makes this run special?
Privacy	PRIVATE
Tags	+
Author	ujjwalx
State	Finished
Job	job-git_git.sr.ht__ujjwal_greenrank_src_mnist_main.py:v16
Start time	November 15th, 2023 at 1:51:05 am
Duration	1h 47m 50s
Run path	ujjwalx/mnist-2/e6tm5qzk
Hostname	mandla-1
OS	Linux-6.2.0-36-generic-x86_64-with-glibc2.35
Python version	3.10.12
Python executable	/home/ujjwal/.cache/pypoetry/virtualenvs/greenrank-llpzgsxI-py3.10/bin/python
Git repository	`git clone git@git.sr.ht:~ujjwal/greenrank`
Git state	`git checkout -b "logical-cherry-29" 90b33156f00a8170ed90b4632fc9d62a04823a8f`
Command	`/home/ujjwal/projects/greenrank/src/mnist/main.py fit --config config.yml`
System Hardware	CPU count 16 GPU count 1 GPU type NVIDIA GeForce RTX 4090
W&B CLI Version	0.15.12

10 Simple Rules for Reproducible Computational Research

1. For Every Result, Keep Track of How It Was Produced
2. Avoid Manual Data Manipulation Steps
3. Archive the Exact Versions of All External Programs Used
4. Version Control All Custom Scripts
5. Record All Intermediate Results, When Possible in Standardized Formats
6. For Analyses That Include Randomness, Note Underlying Random Seeds
7. Always Store Raw Data behind Plots
8. Generate Hierarchical Analysis Output, Allowing Layers of Increasing Detail to Be Inspected
9. Connect Textual Statements to Underlying Results
10. Provide Public Access to Scripts, Runs, and Results

Record Everything

Automate Everything

Sandve GK, Nekrutenko A, Taylor J, Hovig E (2013) Ten Simple Rules for Reproducible Computational Research. PLoS Comput Biol 9(10): e1003285. doi:10.1371/journal.pcbi.1003285

Log your experiments and runs.

Strictly version code and artifacts.

Try to pin down sources of variation and randomness