Efficient Deep Learning Systems
Experiment management & testing

Max Ryabinin

2022

Plan for today

- How (and why) to track your DL experiments
- Versioning your data and models along with the code
- Flexible configuration of Python code
- Testing in general and for ML purposes

Tracking experiments: motivation

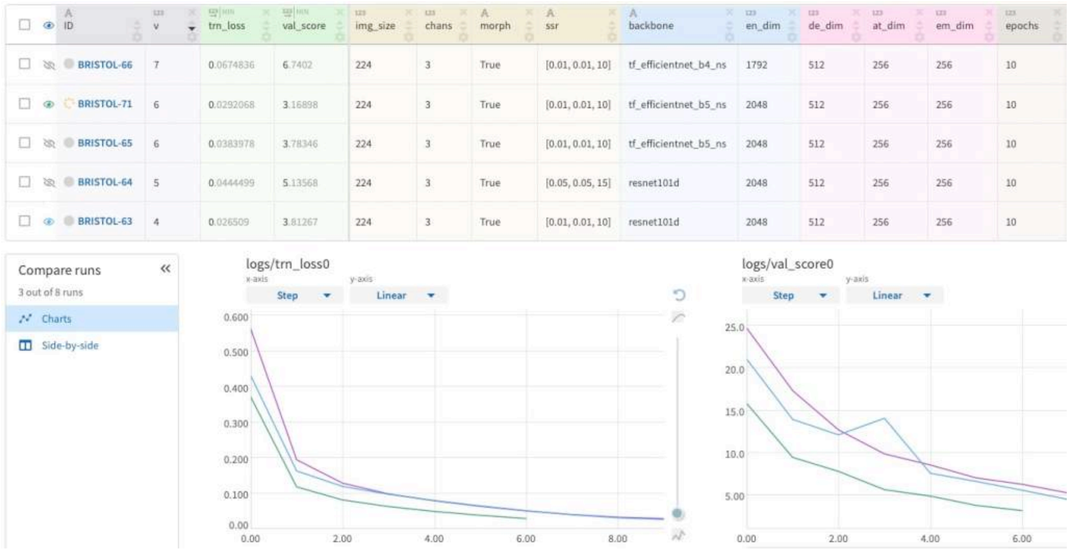
- Usually, training a model once is not enough:
  - The data gets updated
  - Hyperparameters need tuning
  - We want to modify the training code for better quality
- For all these cases, we need a way to keep track of our experiments
- Even more important in a collaborative setup

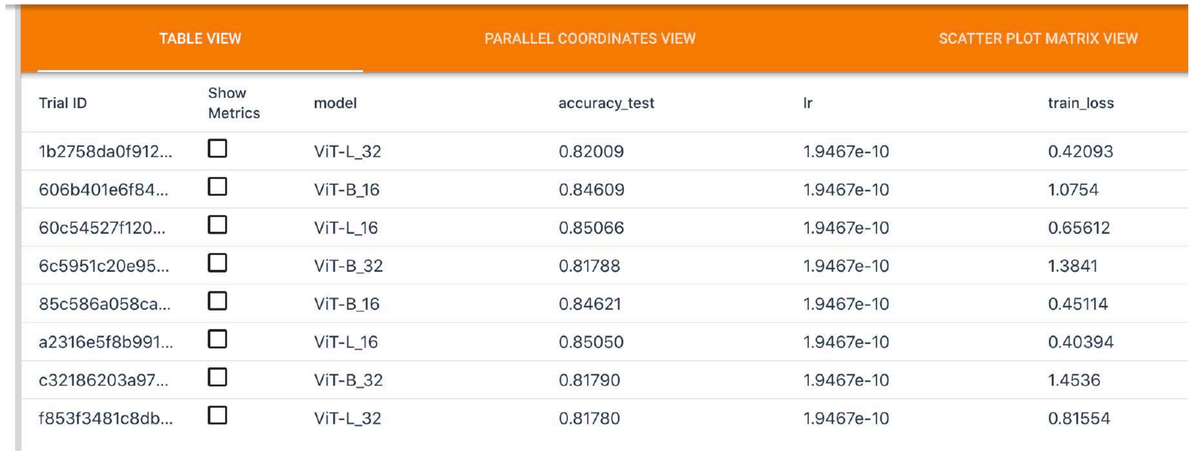
What to track

- Obviously, we want a table with run IDs and final metrics
- What else?
  - Plots with per-step/per-second metrics (convergence & perf)
  - Git commit hash for reproducibility
  - Visualizations of model inputs/outputs
  - In some cases, a full copy of the environment

How to track

- There are many tools for this [1,2,3,4]
- Range from “just upload the logs” to full-fledged tracking of the entire environment
- Self-hosted versions are available





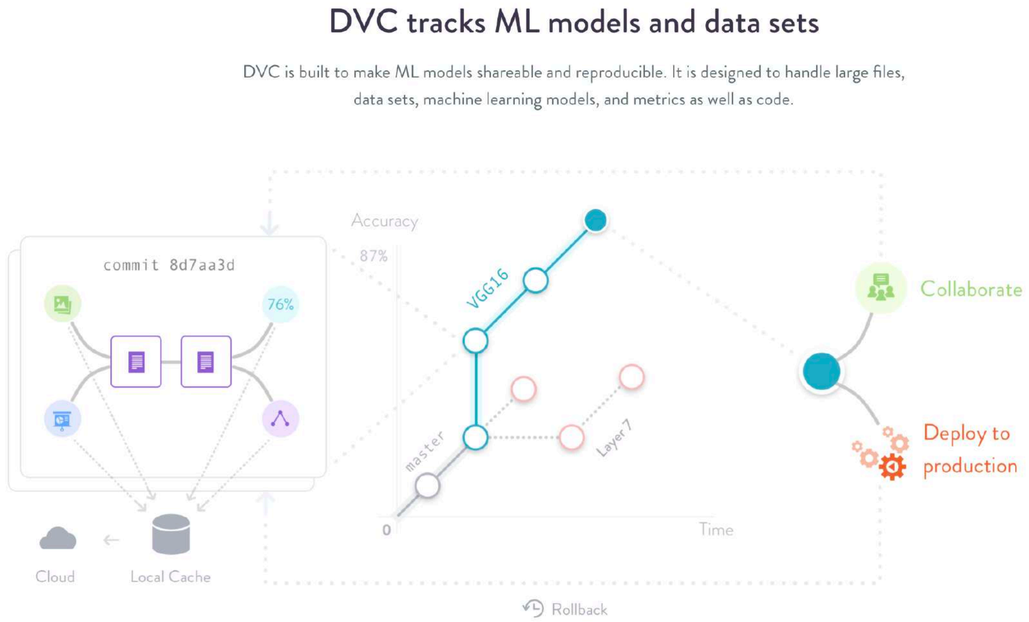
[1] https://www.wandb.com/
[2] https://www.comet.ml/
[3] https://neptune.ai/
[4] https://tensorboard.dev/

Data versioning

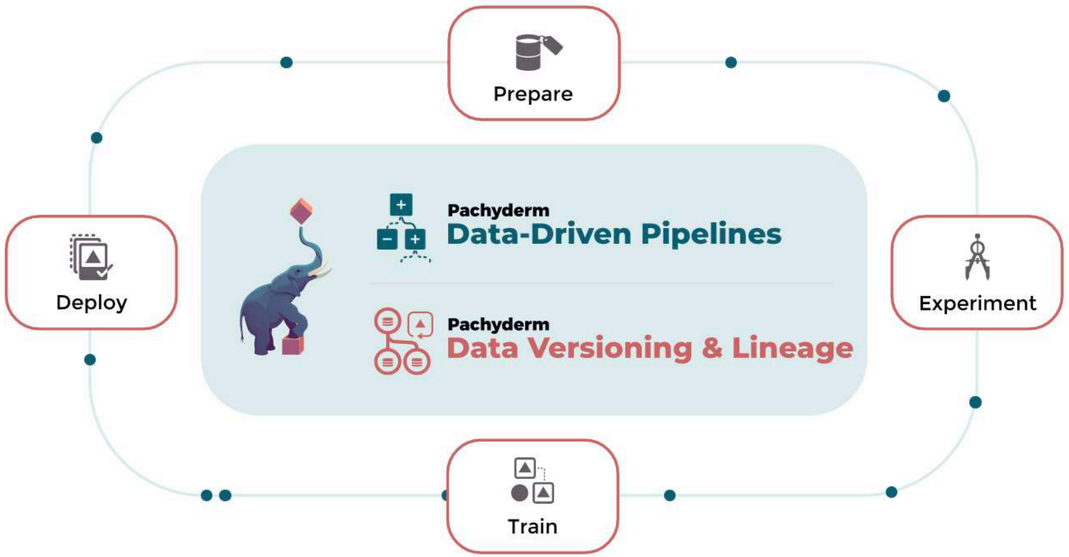
- Code is not the only component in your system
- Data is a crucial dependency, especially in complicated pipelines
- Tracking changes in it is equally important
- Pinning each experiment to its data enhances reproducibility

Solutions

- Several existing projects allow to integrate artifact versioning into pipelines
- Support external storage, matching with commits, metric comparison



https://dvc.org/



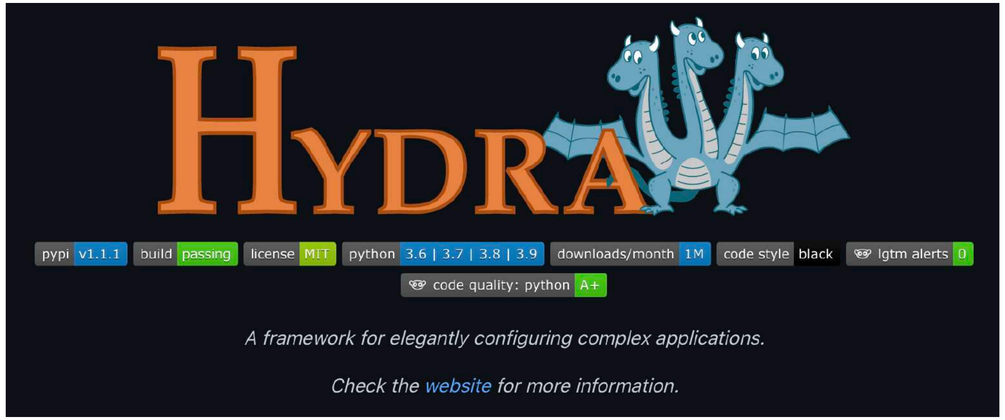
https://www.pachyderm.com/

Configuration

- As your project grows, the number of “moving parts” increases
  - Infrastructure: API endpoints, data URLs, etc.
  - Model hyperparameters and components
- Changing them manually across the entire repo is not sustainable
- Argparse/click-based solutions are hard to write and properly version
- Hardcoding values in dedicated Python files is not flexible enough

Hydra

- One of the most popular solutions for handling configuration
- Uses YAML configs, allows overriding values from the command line
- Simple type checking via Structured Configs
- Grouped configs offer easy switching between groups of presets



Basic example

Config:

conf/config.yaml

```yaml
db:
  driver: mysql
  user: omry
  pass: secret
```

Application:

my_app.py

```python
import hydra
from omegaconf import DictConfig, OmegaConf

@hydra.main(config_path="conf", config_name="config")
def my_app(cfg : DictConfig) -> None:
    print(OmegaConf.to_yaml(cfg))

if __name__ == "__main__":
    my_app()
```

Testing

- In general, testing refers to verifying the intended code properties:
  - Not only correctness, but also performance, handling inputs, etc.
- Why should we test our code?
  - It helps avoid the bugs (both now and when refactoring)
  - It improves the overall code quality by decoupling
  - Essentially, you get self-documented code for free

Types of software tests

- There are many kinds, e.g.:

  1. **Unit tests** verify the correctness of a single component
  2. **Integration tests** ensure that modules work together
  3. **End-to-end tests** verify that the entire application is correct
  4. **Stress/load/performance** tests check the speed of code under load

- We'll focus on 1 and 2: they are the easiest to write and cover most cases

https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing
https://hackr.io/blog/types-of-software-testing

How to test Python code

- Python built-in: unittest
  - Quite simple, ready to use
  - Cons: has its own syntax, not that flexible
- Better: pytest
  - Flexible, works with assert statements, has plenty of integrations via plugins

```python
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

```python
# content of test_sample.py
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                        [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================== short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
============================== 1 failed in 0.12s ===============================
```

Property-based testing

- How do we generate test cases?
  - Coming up with our own inputs is not exhaustive
  - Basically, we only test that the code works for given inputs
  - Furthermore, our requirements become unclear
- Property-based testing aims to solve this problem
  - Instead of specifying exact inputs, we tell what they should be
  - The framework tests the code on many inputs and tries to simplify failing cases

Hypothesis

- A Python framework for property-based-testing
- Integrates with pytest
- Has strategies for generating NumPy arrays (which generalizes to PyTorch tensors)

```python
from hypothesis import given, strategies as st


@given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert x + y == y + x


@given(x=st.integers(), y=st.integers())
def test_ints_cancel(x, y):
    assert (x + y) - y == x


@given(st.lists(st.integers()))
def test_reversing_twice_gives_same_list(xs):
    # This will generate lists of arbitrary length (usually between 0 and
    # 100 elements) whose elements are integers.
    ys = list(xs)
    ys.reverse()
    ys.reverse()
    assert xs == ys


@given(st.tuples(st.booleans(), st.text()))
def test_look_tuples_work_too(t):
    # A tuple is generated as the one you provided, with the corresponding
    # types in those positions.
    assert len(t) == 2
    assert isinstance(t[0], bool)
    assert isinstance(t[1], str)
```

```
>>> import numpy as np
>>> from hypothesis.strategies import floats
>>> arrays(np.float, 3, elements=floats(0, 1)).example()
array([ 0.88974794,  0.77387938,  0.1977879 ])
```