Designing tests for ML libraries – lessons from the wild

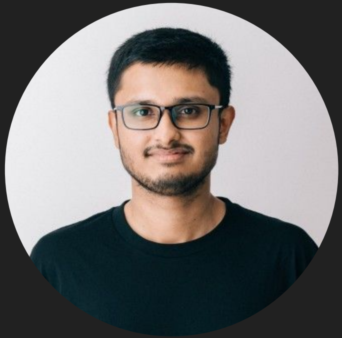
Benjamin Bossan & Sayak Paul
Hugging Face 🤗

PyData Amsterdam, September 2025 🇳🇱

Introduction

$who

Benjamin is part of the `PEFT` team at Hugging Face. He is working on enabling the training of large models on modest hardware.



Sayak is the part of the `Diffusers` team at Hugging Face. He’s involved in a number of applied research initiatives in the area of image and video generation.

Outline

- Revisiting the existing topic of tests 🤷
- A bit about ML libraries and their kinds
- Approaching tests for the OSS libraries at 🤗
- Practical concerns and how we address them
    - With concrete examples and links to the code
- Miscellaneous topics related to testing

Revisiting tests

ML libraries differ from regular Python libraries, which impacts testing

- Data dependency
- Compute and memory bottlenecks
- Non-deterministic outcomes
- Hardware and accelerator dependency

The big 🐘 – ML libraries

- Platform-level (`torch`, `jax`)
- Modeling (`transformers`, `diffusers`)
- Utility (`peft`, `trl`, `axolotl`)
- Data and IO (`torchvision`, `webdataset`, `datasets`)

We discuss `Diffusers` and `PEFT` in this talk 🤗

🤗 Diffusers and PEFT

- 🤗 Diffusers provides a unified interface to work with state-of-the-art open models for diffusion-based {image, video, audio}-generation.
- 🤗 PEFT implements numerous methods for parameter-efficient fine-tuning of PyTorch models on modest hardware.
- Together, 10M monthly PyPI downloads, 50k GitHub stars

The 🤗 approach to tests

Library functionality drives test design

- Tests differ from library to library, depending on the type
- For a modeling library like `Diffusers`, it’s generally a collection of implementations of SoTA models:

Library functionality drives test design

- Tests differ from library to library, depending on the type
- For a modeling library like `Diffusers`, it’s generally a collection of implementations of SoTA models:
  - Quantization support
  - Gradient checkpointing support
  - Fast weight loading
  - Parallelism support
  - Parameter offloading
  - and more …

Library functionality drives test design

- Tests differ from library to library, depending on the type
- For a utility library like `PEFT`, it’s generally a collection of different methods that are tested for functionality:

Library functionality drives test design

- Tests differ from library to library, depending on the type
- For a utility library like `PEFT`, it's generally a collection of different methods that are tested for functionality:
  - Training with these methods works
  - Methods work with different model types (decoder only, encoder-decoder, …)
  - Methods support full range of `PEFT` API (saving/loading, merging, …)
  - Quantization
  - …

Library functionality drives test design

Common tests shared by the model implementations shipped in the library

```python
def test_enable_disable_gradient_checkpointing(...):
    ...
def test_effective_gradient_checkpointing(...):
    ...
def test_gradient_checkpointing_is_applied(...):
    ...
def test_deprecated_kwargs(...):
    ...
def test_save_load_lora_adapter(...):
    ...
def test_lora_wrong_adapter_name_raises_error(...):
    ...
def test_lora_adapter_metadata_is_loaded_correctly(...):
    ...
def test_lora_adapter_wrong_metadata_raises_error(...):
    ...
def test_cpu_offload(...):
    ...
```

[Test File]

Dealing with slow tests

- Testing pre-trained models is slow
  - Long-running test suites on PRs are a bad experience
  - Keep model configurations *minimal yet reasonable*
  - Small variants stored on HF Hub
- Tiers of frequencies in which tests are run:
  - FAST: runs on PRs, merges/pushes to main, uses small models (e.g., `peft-internal-testing/tiny-dummy-qwen2`)
  - SLOW: run on merges/pushes to main (can involve real, pre-trained models)
  - NIGHTLY: run on a nightly basis (real models, usually integration & accelerator tests)

Test matrices

- Ever growing set of models, methods, and features is supported
- Set up a base class of tests for every feature and parametrize it over models and methods
- Extension only requires adding an item to the parameter list

Test matrices

Simplified example from `PEFT` tests:

```python
PEFT_DECODER_MODELS_TO_TEST = [
    "hf-internal-testing/tiny-random-OPTForCausalLM",
    "hf-internal-testing/tiny-random-Gemma3ForCausalLM",
]
ALL_CONFIGS = [
    (LoraConfig, {"task_type": "CAUSAL_LM", "r": 8}),
    (AdaLoraConfig, {"task_type": "CAUSAL_LM", "total_step": 1}),
]

class TestDecoderModels(PeftCommonTester):
    transformers_class = AutoModelForCausalLM

    @pytest.mark.parametrize("model_id", PEFT_DECODER_MODELS_TO_TEST)
    @pytest.mark.parametrize("config_cls,config_kwargs", ALL_CONFIGS)
    def test_save_pretrained(self, model_id, config_cls, config_kwargs):
        self._test_save_pretrained(model_id, config_cls, config_kwargs)
```

Practical concerns & how we address them

Python versions 🐍

- Test current Python versions through the test matrix
- Avoid accidentally using Python features not supported by older versions
- Some dependencies can clash with certain Python versions

Other package versions 📦

- For critical dependencies like 🤗 `Transformers`, test latest source install
- Allows to catch breaking changes early
- There are specific tests for integrations (e.g., `PEFT` ⇔ `Transformers`)

Operating systems 💻

- 3 major platforms: Linux, Windows, MacOS
- PyTorch support for these platforms can differ, so testing is necessary
- MacOS-specific issues
- For `PEFT` and `Diffusers`, we also host our own runners (with GPU) but only for Linux

Concerns with large test matrices

- When testing 4 Python versions and 3 OSes, we already have 12 combinations
- Even though they can run in parallel, we have to wait for the slowest to finish
- If there are flaky tests, it’s more likely that at least one run triggers it
- Leads to a 12x increase in network requests, which can result in rate limits (discussed later)

Benchmark tests 📊

- Speed of ML models is an important concern
- Changes to the modeling code can improve/degrade speed
- We expect the baseline latency NOT to increase with changes to modeling code
- Hence the importance of benchmarking the model runtime
- This also helps improve user trust

Benchmark tests 📊

Considerations

- Use real model checkpoints if the test infrastructure allows it
  - If not, keep the model configuration sufficiently heavy so that numbers are realistic
- If parameter offloading is used, benchmark forward passes with offloading enabled
- Only test most popular models if load is too heavy
- Thorough reporting of benchmarking results
  - Track how latency and other speed-related metrics are changing over time
  - Flag weird changes if needed

Benchmark tests 📊

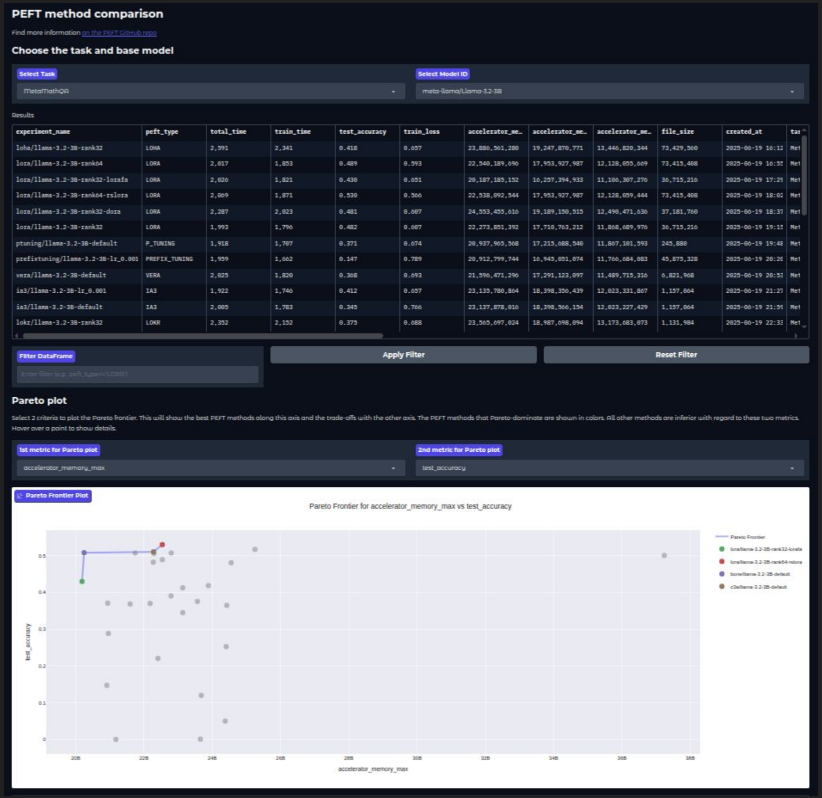
num_params_B	flops_G	time_plain_s	mem_plain_GB	time_compile_s	mem_compile_GB	fullgraph	mode	github_sha
11.90	59523.04 (59529.52)	0.55 (0.544)	22.63 (22.64)	0.381 (0.405)	22.73	True	default	20e0740b882678353461455facc682494a493775
5.95	NaN	0.566 (0.577)	6.72	NaN	NaN	NaN	NaN	20e0740b882678353461455facc682494a493775
11.90	59523.04 (59529.52)	0.604	22.18	NaN	NaN	NaN	NaN	20e0740b882678353461455facc682494a493775
11.90	59523.04 (59529.52)	1.897 (1.9)	0.55	NaN	NaN	NaN	NaN	20e0740b882678353461455facc682494a493775
13.04	167565.8 (167583.45)	1.668 (1.638)	25.22	1.131 (1.143)	25.31	True	default	20e0740b882678353461455facc682494a493775

`Diffusers` benchmarking workflow

Benchmark tests 📊

[PEFT benchmark]()

Compare different `PEFT` methods on the same task and measure performance, runtime, memory, etc.



Code coverage

```
Name                                             Stmts   Miss  Cover   Missing
----------------------------------------------------------------------------------
src/peft/__init__.py                                10      0   100%
src/peft/auto.py                                    71      4    94%   61, 92, 99, 111
src/peft/config.py                                 133      5    96%   89, 156, 242, 267-268
src/peft/helpers.py                                 72     21    71%   48-58, 84-98, 124-132, 235
[...]
src/peft/utils/warning.py                            1      0   100%
----------------------------------------------------------------------------------
TOTAL                                            17430   4540    74%
```

Code coverage

- High code coverage is generally desirable, coverage metrics can reveal gaps in tests
- Easily added via `pytest-cov`
- Due to stratification, there is no single place where all tests are run
- Monitoring test coverage is crucial for large refactors (especially refactors of the test suite itself)
- Experimental features can be added without tests at first

Regression testing

- Check that model outputs remain constant over time
- This is crucial for users, who expect backwards compatibility
- We use Hugging Face Hub to store regression artifacts

Regression testing

```python
class TestModelRegression(RegressionTester):
    def load_base_model(self):
        self.fix_seed()
        return AutoModelForCausalLM.from_pretrained(...)

    def test_lora(self):
        base_model = self.load_base_model()
        config = LoraConfig(
            r=8,
            init_lora_weights=False,
        )
        model = get_peft_model(base_model, config)
        self.assert_results_equal_or_store(model, <name>)

```

[PEFT regression tests]

Dealing with known failures

- ML library tests can fail in different ways:

Dealing with known failures

- ML library tests can fail in different ways:
  - Platform level (kernel bug, operator bug)
  - Python versions
  - Bugs from any required dependency

Dealing with known failures

- ML library tests can fail in different ways:
  - Platform level (kernel bug, operator bug)
  - Python versions
  - Bugs from any required dependency
- Rather than skipping, prefer using `pytest.mark.xfail` and supplement a reason and a condition
  - A better way to communicate about known failures and when they are likely to happen

Dealing with known failures

```python
@pytest.mark.xfail(
    condition=torch.device(torch_device).type == "cpu"
    and is_torch_version(">=", "2.5"),
    reason="Test currently fails on CPU and PyTorch 2.5.1 but not on PyTorch 2.4.1.",
    strict=False,
)
def test_lora_fuse_nan(self):
    ...
```

Determinism and flakiness

- Non-determinism – outputs changing every time a model/method is run on same inputs

Determinism and flakiness

- Non-determinism – outputs changing every time a model/method is run on same inputs
- Many sources:
  - Stochastic operations inside models (instances of `torch.randn`, for example)
  - RNG seed not controlled properly (PyTorch’s RNG is an advanced one)
  - Use of non-deterministic algorithms (for performance reasons)

Determinism and flakiness

- Non-determinism – outputs changing every time a model/method is run on same inputs
- Many sources:
    - Stochastic operations inside models (instances of `torch.randn`, for example)
    - RNG seed not controlled properly (PyTorch's RNG is an advanced one)
    - Use of non-deterministic algorithms (for performance reasons)
- Tests pass after a certain number of retries (can be very unreasonable sometimes) – flaky
    - Tests may pass locally in the first go
    - Tests may pass after several retries in the CI

Determinism and flakiness

- We disable non-determinism at the PyTorch level for tests where it's possible to produce non-deterministic results

```python
def enable_full_determinism():
    """
    Helper function for reproducible behavior during distributed training. See
    - https://pytorch.org/docs/stable/notes/randomness.html for pytorch
    """
    os.environ["CUDA_LAUNCH_BLOCKING"] = "1"
    os.environ["CUBLAS_WORKSPACE_CONFIG"] = ":16:8"
    torch.use_deterministic_algorithms(True)

    # Enable CUDNN deterministic mode
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.backends.cuda.matmul.allow_tf32 = False
```

Determinism and flakiness

For models with stochastic operations, make them accept an RNG to control that behaviour

```diff
 class MyModel(torch.nn.Module):
     def __init__(self, ...):
         ...

-    def forward(self, ...):
-        ... = torch.randn(...)
+    def forward(self, ..., generator=None):
+        ... = torch.randn(..., generator=generator)
```

Determinism and flakiness

When the model needs to be invoked multiple times, re-initialize the generator each time

```
def test_function(model):
    ... = model(..., generator=torch.manual_seed(0))

    ... = model(..., generator=torch.manual_seed(0))
```

Determinism and flakiness

To relax assertion in tests, consider using thresholds with similarity metrics (such as cosine similarity)

```python
# Say, a tensor of shape (1, 3, 32, 32)
generated_output = ...
generated_output generated_output.flatten()
generated_slices = torch.cat(
  [generated_output[:8], generated_output[-8:]]
)

assert cosine_similarity_distance(generated_slices, expected_slices) < 1.0
```

Determinism and flakiness

For flaky tests mark them as such, setting maximum retries

```python
@is_flaky(max_attempts=10, description="very flaky class")
class WanVACELoRATests(unittest.TestCase, PeftLoraLoaderMixinTests):
    ...
```

Caching of models and datasets

- Often, model and dataset artifacts need to be downloaded
- Caching reduces network traffic, speeds up tests, and reduces issues like rate limits
- Session-level caching can be implemented via the testing framework (e.g. session-scoped fixtures in `pytest`)
- Even if a HF model is cached locally, calling `AutoModel.from_pretrained` sends a request to check for updates
    - Use `local_files_only=True` or `HF_HUB_OFFLINE=1` to avoid this
- It's possible to use GitHub cache to avoid downloading all the artifacts from HF Hub

Caching of outputs

- Consider caching outputs that are reused in tests
- A good example of this is the `Diffusers`-LoRA testing suite
- Leverage `@pytest.fixture` with a class scope for this

Miscellaneous

PyTorch-specific considerations

- Use `torch.allclose or torch.testing.assert_close` for tensors
- Use [`skip` markers] to omit tests that require specific hardware
- Remember to set seeds before the test starts and to free memory after the test finishes
  - Can be automated with pytest fixtures: `@pytest.fixture(autouse=True)`

Checking code quality

- We enforce uniformity in terms of how code is styled and run basic QC on it
- Our preferred tool for this is `ruff` because because of its speed
- Especially important as we have contributions from many outside contributors with very different backgrounds

Improving test runtime

When tests are slow:

- Identify the tests that take the most amount of time (`pytest --durations` reports are a great tool for this)
- If possible, shrink the models being used for the tests, cache outputs, etc.
- Move slow but non-essential tests to a different schedule (e.g. only run them nightly, not per PR)
- More powerful machines can run tests faster, but the first option should be preferred as it's less costly

Testing framework

- The two big testing frameworks are `unittest` (built-in) and `pytest`
- Both are very capable and used by Hugging Face
- As `pytest` is widely adopted and provides many creature comforts, we tend towards migrating to it
- We always use `pytest` as a test runner because of its many useful features:

Testing framework

- The two big testing frameworks are `unittest` (built-in) and `pytest`
- Both are very capable and used by Hugging Face
- As `pytest` is widely adopted and provides many creature comforts, we tend towards migrating to it
- We always use `pytest` as a test runner because of its many useful features:
  - Useful options to run only subsets of tests (`pytest -m`, `pytest -k`)
  - Exit on first error (`-x`) or drop into a debugger (`--pdb`)
  - Run tests in parallel with `pytest-xdist`
  - And many more

Community contributions

- Make it easy to set up testing for all contributors (helps both maintainers and contributors)
- Make PR tests run fast without compromising library integration
- Lenient when it comes to code duplication in tests

Community contributions

- Make it easy to set up testing for all contributors (helps both maintainers and contributors)
- Make PR tests run fast without compromising library integration
- Lenient when it comes to code duplication in tests
- Use automated bots when needed

Conclusion

- Tests for ML libraries can differ in many aspects from regular libraries
- Testing should be a first class concern of your project
- Tests should be wide to cover all use cases, a select amount of tests should be deep to verify the details
- When facing constraints, prioritize and stratify the tests
- Use the breadth of your tools to make life easier

Resources

- Slides: bit.ly/hf-tests
- `Diffusers`: github.com/huggingface/diffusers
- `PEFT`: github.com/huggingface/peft
- `pytest` guide: github.com/BenjaminBossan/pytest-guide