The CERN ML Frameworks

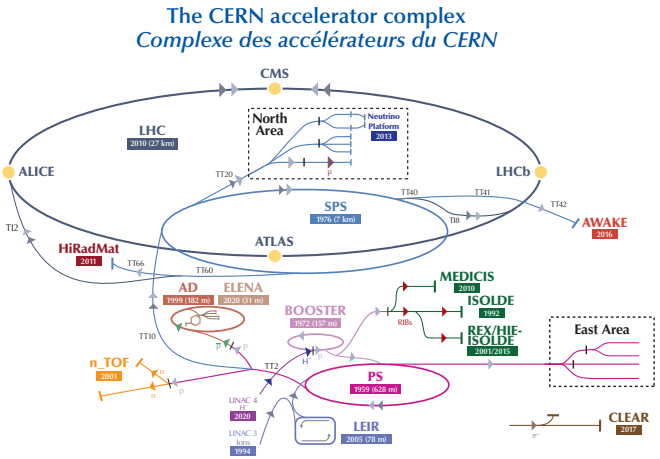
How to get ML into the Control Room

J.-B. de Martel ▪ N. Madysa R. Gorbonosov V. Kain

3rd ICFA Beam Dynamics Mini-Workshop,
on Machine Learning Applications for Particle Accelerators,
3 November 2022

CERN

- many accelerators, extremely diverse
- lots of low-level problems already well automated
- but: many high-level problems still solved manually
- better turnaround time and beam quality necessary to reach target integrated luminosity



CERN

Lots of infrastructure already in place!

- JAPC: uniform communication protocol with physical and virtual devices
- LSA: access to the settings database and history
- UCAP: *Function as a Service* development framework (Link)
- WRAP: web app to create browser-based monitoring dashboards (Link)
- Acc-Py: Python development and deployment framework

Two projects are more specialized on optimization and ML:

- MLP: platform for ML model versioning, storage and deployment
- GeOFF: Framework for operational use of RL and numerical optimization

Machine Learning Platform (MLP)

Volatile world of ML:

- code needs to run once
- bleeding-edge technology
- cloud services and custom tools
- maintainability not the first concern

Reliable world of accelerator controls:

- need to run reliably 24/7/365: reproducibility, robustness, traceability
- highly reliable, battle-tested tools
- constraints of the technical network: no internet access, restricted tooling, security precautions
- minimize maintenance effort through standardization and unification

Machine Learning Platform (MLP)

Objectives:

- common approach to storage, versioning, deployment and usage of models
- remove infrastructural concerns from model lifecycle
- fulfill needs specific to accelerator controls:
  - reliability
  - traceability
  - security
  - standardization
- stay out of the user's way:
  - minimize constraints on model developer's workflow
  - don't restrict choice of tools

Machine Learning Platform (MLP)

Concepts:

Model Type

- contains the model code
- fully fledged Python package
- lives as a Git repository on our servers
- different versions are different development snapshots

Model Parameters

- contains only the training results
- depends on the ML framework
- often an opaque binary file
- different versions are different training snapshots

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package

```
$ tree
my-model-package/
├── my_model_package/
│   ├── __init__.py
│   └── mlp-models.toml
├── .gitlab-ci.yaml
├── pyproject.toml
└── setup.cfg
```

Made easy with **`acc-py init`** and **`acc-py init-ci`**!

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package
2. Declare your model

```toml
# mlp-models.toml
[[model]]
name = "my_model_package:MyModel"
standalone = false
```

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package
2. Declare your model
3. Implement a wrapper class

```python
# __init__.py
from mlp_model_api import INPUTS, OUTPUTS
from mlp_tensorflow_model \
    import TensorflowModel
class MyModel(TensorflowModel):
    def __init__(self):
        model = tf.keras.Model(...)
        super().__init__(model)
    def predict(self, inputs):
        outputs = self.model(\
            inputs[INPUTS])
        return {OUTPUTS: outputs}
```

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package
2. Declare your model
3. Implement a wrapper class
4. optional: other frameworks

```python
# __init__.py
class MyModel(TensorflowModel):
    ...
    def export_parameters(
        self, parameters_target: Path
    ) -> None:
        ...
    def load_parameters(
        self, parameters_src Path
    ) -> None:
        ...
```

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package
2. Declare your model
3. Implement a wrapper class
4. optional: other frameworks
5. Push a Git tag to CI

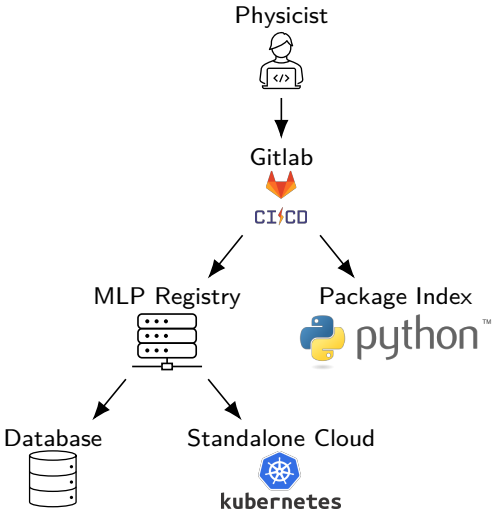
```
$ git commit --all -m "New version."
$ git push origin main
$ git tag v1.0.0
$ git push --tags origin
```

Machine Learning Platform (MLP)

Developing a model:

1. Set up your package
2. Declare your model
3. Implement a wrapper class
4. optional: other frameworks
5. Push a Git tag to CI

... and CI takes care of the rest!



Machine Learning Platform (MLP)

Publishing training results to the MLP:

```python
# train_my_model.py
from my_model_package import MyModel
from mlp_client import AUTO, Client, Profile

mlp_model = MyModel()
train(mlp_model.model)
client = Client(Profile.PRO)
client.publish_model_parameters_version(
    mlp_model, name="my_model.default", version=AUTO
)
```

Machine Learning Platform (MLP)

Embedding an MLP model in a Python application:

```python
# operational_application.py
from my_model_package import MyModel
from mlp_client import AUTO, Client, Profile
from mlp_model_api import INPUTS, OUTPUTS

client = Client(Profile.PRO)
model = client.create_model(
    MyModel, name="my_model.default", version=AUTO
)
response = model.predict({INPUTS: get_inputs()})
show_results(response[OUTPUTS])
```

Machine Learning Platform (MLP)

Connecting a Python application to a stand-alone model in the cloud:

```python
# operational_application.py
from mlp_client import AUTO, Client, Profile
from mlp_model_api import INPUTS, OUTPUTS

client = Client(Profile.PRO)
model = client.create_standalone_model(
    "my_model_package:MyModel",
    name="my_model.default",
    version=AUTO,
)
# Communicates via HTTP!
response = model.predict({INPUTS: get_inputs()})
show_results(response[OUTPUTS])
```

Machine Learning Platform (MLP)

Connecting a Java application to a stand-alone model in the cloud:

```java
// OperationalApplication.java
public static void main() {
    StandaloneModel<MyRequest, MyResponse> model =
        new MlpClient(Environment.PRO)
        .standaloneModelBuilder(MyRequest.class, MyResponse.class)
        .modelType("my_model_package:MyModel")
        .modelParameters("my_model.default")
        .build();
    MyRequest request = new MyRequest(getInputs());
    MyResponse response = model.predict(request);
    showResults(response.getOutput());
}
```

Machine Learning Platform (MLP)

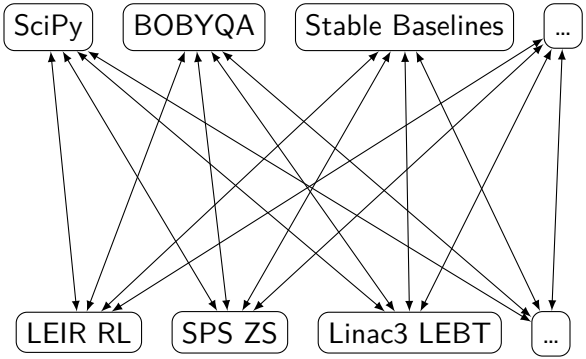
Conclusion:

- MLP is ready for production*
- we separate model R&D from its operational use
- model developers can ignore infrastructural concerns
- we follow software engineering best practices
  $\Rightarrow$ reliable and maintainable

**standalone model deployment in beta*

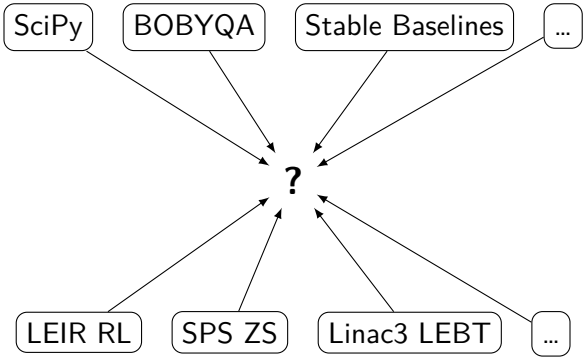
GeOFF

- many different optimizers/APIs
- many different optimization problems
- each problem involves complex machine communication
- operators don't want to juggle Python scripts!



GeOFF

- many different optimizers/APIs
- many different optimization problems
- each problem involves complex machine communication
- operators don't want to juggle Python scripts!



GeOFF

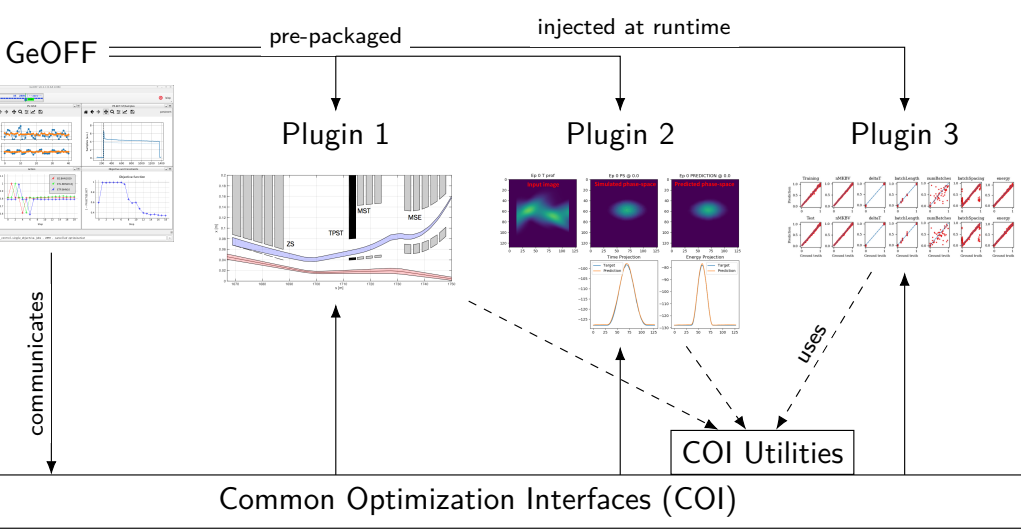
Goals:

- provide ecosystem for accelerator optimization and control
- provide compatibility with as many algorithms as possible
- facilitate the progression
  manual tuning $\rightarrow$ *numerical optimization* $\rightarrow$ *machine learning*

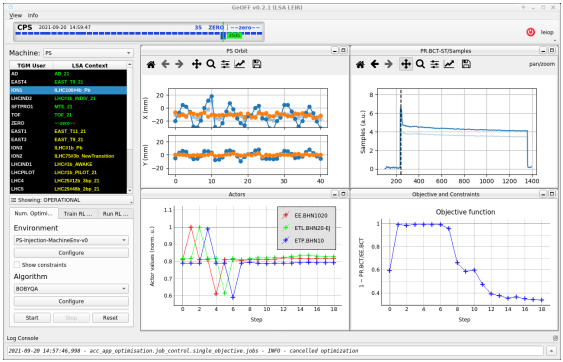
Guiding principles:

- be agnostic over machine, communication protocol or devices
- minimize boilerplate code that does not solve the problem
- don't make people pay for features they don't use
- always leave an escape hatch open

GeOFF


GeOFF
pre-packaged
injected at runtime
Plugin 1
Plugin 2
Plugin 3
communicates
uses
COI Utilities
Common Optimization Interfaces (COI)

GeOFF



- lists, configures and runs optimization problems
- built-in list of optimizers (will be made user-extensible)
- optimization problems are loaded as plugins pre-packaged or at runtime

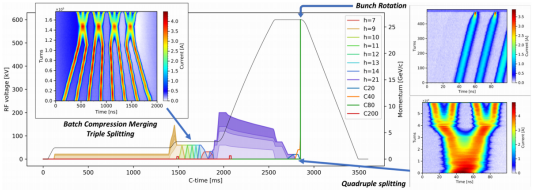
GeOFF Use Cases

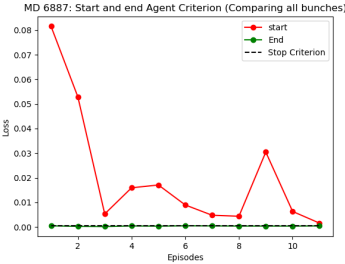
- Linac3: steering of beam transfer line
- Linac4: 2 expert tools
- PSB: operations (WIP) & commissioning
  - bunch recombination at PSB ejection
  - resonance compensation
  - RF optimization
  - injection to PS
- PS: used during commissioning
  - resonance compensation
  - transition gamma jump circuits
  - septa alignment for slow extraction
- LEIR: used during commissioning
  - transfer lines (from Linac3, to PS)
  - injection bumps
  - phase adjustment of RF cavities
- SPS: expert tool & operations
  - tune adjustments
  - septa alignment for slow extraction
  - spill noise reduction
  - splitter optimization
  - injection kicker optimization
  - crystal shadowing
- used at almost all accelerators
  - ISOLDE: lots of homogeneous devices
    $\Rightarrow$ CPS Optimizer
  - LHC: fast acquisition, high safety req.
    $\Rightarrow$ bespoke algorithms
- most often used as expert tool

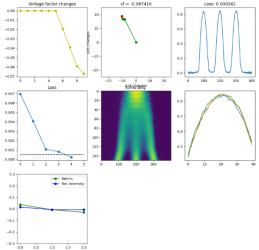
Use Case: PS RF Manipulations via RL

- **LHC beam production** requires quadruple splitting at $26\,\mathrm{GeV}/c$ in PS
- **RF phase errors** introduce spread in bunch-by-bunch intensity and emittance

$\Rightarrow$ RL agent corrects phases for uniform bunches







- faster than classical optimization due to reuse of experience
- trained on simulation, evaluated on real machine
- episode length $n \in [2, 18]$, $\bar{n} = 8.46$

presented yesterday by J. Wulff

Conclusion

- CERN fosters an ecosystem of independent efforts to improve operational ML usage
- solutions complement and integrate with each other
- but: still plenty to do!
- efforts to open-source MLP and GeOFF are underway
  - upcoming EURO-LABS project will drive this effort further

Links:

- https://gitlab.cern.ch/nmadysa/icfa-tutorial
- https://cernml-coi.docs.cern.ch/
- https://gitlab.cern.ch/be-op-ml-optimization
- https://indico.cern.ch/event/1175862/ (seminar on MLP and GeOFF)