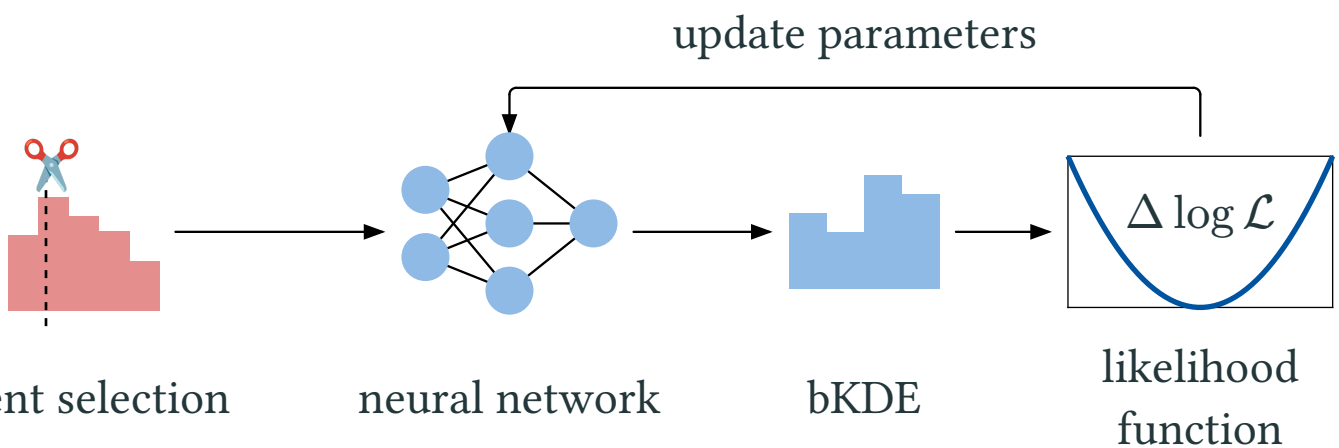
ParaO – A User Perspective

First hand experience within a Run 3 analysis

Benjamin Fischer, **Felix Zinn**

PyHEP.dev 2026 – Nikhef, Amsterdam

Use Case: Differentiable Analysis



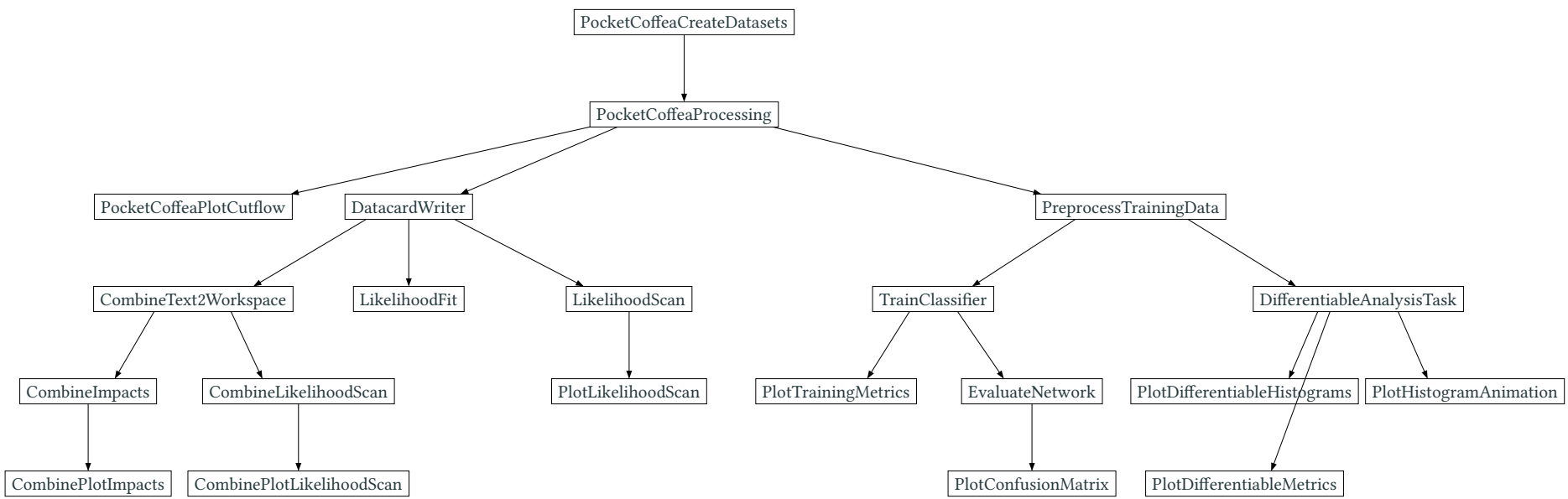
- differentiable analysis workflow
- compare with established methods

Tasks in the Workflow

1. Processing of event files for pre-selection (PocketCoffea)
2. Preprocessing for neural network
3. Neural network training/evaluation
   - classifier
   - systematics aware
4. statistical analysis
   - evermore and comparison with CMS combine
   - goodness-of-fit
   - likelihood scan
   - pulls/impacts
   - ...
5. Plotting for all tasks

$\longrightarrow$ 21 Tasks

Tasks in the Workflow



Overview

Setup Tasks

```py
class MyTask(Task):
  @RunAction
  def run(self) -> Dir:
    # do stuff
    return
```

Execute Tasks

```py
cli = CLI({
  (MyTask, "param"): "value",
})

cli.run
```

Configure (not only) Tasks

```py
class Optimizer(ParaO):
  name = Param[str]()
  learning_rate = Param[float]()
  momentum = Param[float]()

```

For internal details about ParaO see the previous talk at PyHEP.dev 2026 (Benjamin Fischer)
ParaO: Ease of Use for In Situ Workflows

GitHub | GitLab

Tasks

```py
from parao import Dir, Param, RunAction, Task

class TrainClassifier(Task):

  @RunAction
  def run(self) -> Dir:
    output_dir = self.run.output.temp()
    # do the training
    return output_dir
```

- output directory to store all outputs of the task
- temporary directory during the run
  - moved after successful completion
  - deleted after failure

Tasks

```py
from parao import Dir, Param, RunAction, Task

class TrainClassifier(Task):
  # task dependency
  training_data = Param[PreprocessTrainingData]()
  # other parameters
  epochs = Param[int](label=True, pos=0)
  batch_size = Param[int](label=True, pos=1)
  backend = Param[str](default="cpu", significant=False)

  @RunAction
  def run(self) -> Dir:
    output_dir = self.run.output.temp()
    # do the training
    return output_dir
```

Tasks

```py
from parao import Dir, Param, RunAction, Task

class TrainClassifier(Task):
  # task dependency
  training_data = Param[PreprocessTrainingData]()
  # other parameters
  epochs = Param[int](label=True, pos=0)
  batch_size = Param[int](label=True, pos=1)
  backend = Param[str](default="cpu", significant=False)

```

```
TrainClassifier(batch_size=500, epochs=10, training_data=PreprocessTrainingData()): missing
└─PreprocessTrainingData(): missing
```

```
    # do the training
    return output_dir
```

Recursive Actions

- define a `BaseTask` for `Actions` that should affect all Tasks
- `Actions`: method on the task (and its dependencies) that does something
- used as a decorator

```py
class BaseTask(Task):
  @RecursiveAction
  def paths(self, depth: int, more: int):
    # custom paths information
    output_path = self.run.output.path
    task_name = self.__class__.__name__
    print(f"{task_name} -> {output_path}")

  @RecursiveAction
  def short_status(self, depth: int, more: int):
    # custom status information
    ...
```

Recursive Actions

→ custom status report with `RecursiveActions`

```sh
python workflow.py TrainClassifier --batch_size=500 --epochs=10 --paths

'TrainClassifier -> output_dir/TrainClassifier/epochs=10/batch_size=500/
run_3ae14e415ca85dae6dd12ebbf09d5f83f4bb09624813e335d19fe60eaff1433b.dir'
└─'PreprocessTrainingData -> output_dir/PreprocessTrainingData/
run_af2138b7a8c33a1fe4934d96fed9f43ca0a9b5459aa95e194293515ae87d15c9.dir'

python workflow.py TrainClassifier --batch_size=500 --epochs=20 --paths
'TrainClassifier -> output_dir/TrainClassifier/epochs=20/batch_size=500/
run_68f63029e67f60cc6cf11349edb15741e992371b6af7d3487ae5fc4f28c9f5dc.dir'
└─'PreprocessTrainingData -> output_dir/PreprocessTrainingData/
run_af2138b7a8c33a1fe4934d96fed9f43ca0a9b5459aa95e194293515ae87d15c9.dir'
```

CLI

Setup Tasks

```py
class MyTask(Task):
  @RunAction
  def run(self) -> Dir:
    # do stuff
    return
```

Execute Tasks

```py
cli = CLI({
  (MyTask, "param"): "value",
})

cli.run
```

Configure (not only) Tasks

```py
class Optimizer(Para0):
  name = Param[str]()
  learning_rate = Param[float]()
  momentum = Param[float]()
```

ParaO CLI

- execute task methods
- define default parameters
- define output template

```py
base_cli = CLI({
  (FancyTemplate, "dir_base"): "/net/scratch/analysis",
  (CreateDatasets, "priority_sites"): ["T2_DE_RWTH"],
  (Processing, "remove"): False,
})
```

```py
from analysis import (
  TrainClassifier,  base_cli
)

if __name__ == "__main__":
  cli = base_cli.sub({
      TrainClassifier: Args(
        epochs=[10, 20],
        batch_size=[500, 1000],
      )
  })
```

1. define `base_cli` for project level parameters
2. use `cli.sub` to define workflow specific configurations

CLI in a Workflow

```py
from parao import Args

from analysis import TrainClassifier, base_cli

if __name__ == "__main__":
  cli = base_cli.sub({
      TrainClassifier: Args(
        epochs=[10, 20],
        batch_size=[500, 1000],
      )
  })
```

- define (default) values for parameters
- expansion to create multiple task configurations

CLI in a Workflow

```py
class TrainClassifier(Task):
  # task dependency
  training_data = Param[PreprocessTrainingData]()
  # other parameters
  epochs = Param[int](label=True, pos=0)
  batch_size = Param[int](label=True, pos=1)
```

```
from parao import Args

from analysis import TrainClassifier, base_

if __name__ == "__main__":
  cli = base_cli.sub({
      TrainClassifier: Args(
        epochs=[10, 20],
        batch_size=[500, 1000],
      )
  })
```

- define (default) values for parameters
- expansion to create multiple task configurations

CLI in a Workflow

```py
from parao import Args

from analysis.tasks import TrainClassifier
from analysis.base import base_cli
```

```
TrainClassifier(batch_size=500, epochs=20, training_data=PreprocessTrainingData()): missing
└─PreprocessTrainingData(): missing

TrainClassifier(batch_size=500, epochs=10, training_data=PreprocessTrainingData()): missing
└─PreprocessTrainingData(): missing
```

```
    )
  })
  cli.run()
```

- define parameter defaults
- use expansion to create multiple task configurations

Configuration

Setup Tasks

```py
class MyTask(Task):
  @RunAction
  def run(self) -> Dir:
    # do stuff
    return
```

Execute Tasks

```py
cli = CLI({
  (MyTask, "param"): "value",
})

cli.run
```

Configure (not only) Tasks

```py
class Optimizer(Para0):
  name = Param[str]()
  learning_rate = Param[float]()
  momentum = Param[float]()

```

Configuration with ParaO

```py
from parao import Param, ParaO

class DNNArchitecture(ParaO):
  layers = Param[list[int]]()
  dropout_rate = Param[float](None)

class DNNOptimizer(ParaO):
  name = Param[str]()
  learning_rate = Param[str]()

class TrainClassifier(ParaO):
  architecture = Param[DNNArchitecture]()
  optimizer = Param[DNNOptimizer]()
```

- dataclass like object
- can be used to define “configuration classes”
- appear in the hash of the task
  - unique and reproducible
  - can use loops and other constructs

Configuration with ParaO

```py
from parao import Param, ParaO

class DNNArchitecture(ParaO):
  layers = Param[list[int]]()
  dropout_rate = Param[float](None)

class DNNOptimizer(ParaO):
  name = Param[str]()
  learning_rate = Param[str]()

class TrainClassifier(ParaO):
  architecture = Param[DNNArchitecture]()
  optimizer = Param[DNNOptimizer]()
```

```py
CLI({
  TrainClassifier: Args(
    architecture=DNNArchitecture(
      layers=[200, 200, 200],
      dropout_rate=[0.2, 0.4, 0.6],
    ),
    optimizer=DNNOptimizer(
      name="Adam",
      learning_rate=0.001,
    ),
  )
})
```

- Architecture and Optimizer configuration is workflow specific
- save in workflow.py and track with git

Configuration (more complex example)

```py
class DNNArchitecture(ParaO):
    layers: list[int] = Param[list[int]]()
    dropout_rate: float = Param[float](None)
    batch_norm = Param[bool]()
    activation = Param[str]()

class DNNOptimizerParams(ParaO):
    name: str = Param[str]()
    init_value = Param[float]()
    peak_value = Param[float](None)
    end_value = Param[float](None)
    warmup_epochs = Param[int](None)
```

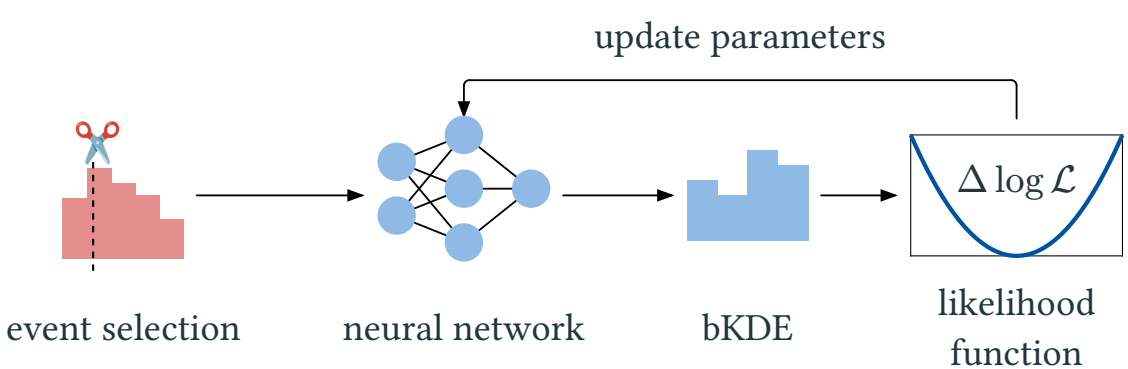
```py
class DNNInputFeature(ParaO):
    name = Param[str]()
    columns = Param[list[str]]()
    max_count: int = Param[int]()

class DNNOutputClass(ParaO):
    name = Param[str]()
    label = Param[int]()
    processes = Param[list[str]]()
```

```py
class Classifier(ParaO):
    dnn_architecture = Param[DNNArchitecture]()
    dnn_optimizer_params = Param[DNNOptimizerParams]()
    input_features = Param[list[DNNInputFeature]]()
    output_classes = Param[list[DNNOutputClass]]()
```

Summary

What I Like

- simple dependency handling
- clean parameter handling
- “set-and-forget” style output templating
- significant parameters affect the output
- configuration using `ParaOs`



GitHub

GitLab

PyPI

Backup

Tasks in my Workflow

- CombineLikelihoodScan
- CombineImpacts
- CombineText2Workspace
- CombinePlotLikelihoodScan
- CombinePlotImpacts
- DifferentiableAnalysisTask
- PlotDifferentiableHistograms
- PlotDifferentiableMetrics
- PlotHistogramAnimation
- EvaluateNetwork
- PlotConfusionMatrix
- PlotTrainingMetrics
- PreprocessTrainingData
- TrainClassifier
- PocketCoffeaCreateDatasets
- PocketCoffeaPlotCutflow
- PocketCoffeaProcessing
- DatacardWriter
- LikelihoodFit
- LikelihoodScan
- PlotLikelihoodScan

Checklist

- [ ] define Tasks with parameters
- [ ] define dependencies through `Param`
- [ ] define default parameters, configuration, output template in CLI
- [ ] add workflow script(s) and track them with git