HALMSTAD UNIVERSITY

# Hands on session

Exploring MlFlow and Optuna in a simple ML-pipeline

# Resources

- Repository: **handson_mlflow_optuna**
- *Git clone https://github.com/caisr-hh/handson_mlflow_optuna.git*
- *MISSING FROM THE ENV: Pytorch!*
- *To install:*

  *Conda activate ml_tooling tutorial*

  *(Linux:)* pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cpu

  *(Windows:)* pip3 install torch torchvision

- Now; step by step examples:
  - Configuration driven development.
  - Hyperparameter optimization.
  - Logging and run management.
  - Deployment.

# Hands on 0:

- The repo is a simple ml pipeline that trains a mlp classifier on a circles dataset.
- Checkout the Tutorial-1 branch and try running it once as is, executing the naked pipeline without optuna or mlflow.

# Hyperparameter selection (Optuna)

Optuna is a library for hyperparameter optimization that attempts to go beyond grid or random searches by modelling the statistical importance of the parameters over several training runs.

We initialize a study that orchestrates this optimization process and define objectives that return the final metric we are optimizing. Over time the trials should sample from a more ideal set of hyperparameter than doing so randomly.

Also has an GUI thanks to optuna-dashboard!

# Studies

A Study is a collection of different objectives being optimized.

```
study = Optuna.create_study(
        str storage=None,           <Storage for trials, if None uses in memory storage.
        pruner=None,                <Optuna pruner if so desired, we will return to this.
        str study_name=None,        <Study name, generated automatically if left empty.
        str direction=None,         <Trying to “minimize” or “maximize” metric?
        bool load_if_exists=False,  <Return existing study with same name?
        [str] directions=None       <List of directions for multiple objectives.
        )
```

Yields an optuna.study.Study object…

# Suggest

For each new trial, the suggest methods invoked inside the objective provides an informed guess as to what parameters influences the objective metrics in the desired directions.

- Trial.suggest_float(string name, float low, float high, float step=None, bool log=False)
- Trial.suggest_int(string name, float low, float high, float step=None, bool log=False)
- Trial.suggest_categorical(string name, float low, float high)

# Studies

…That can be optimized:

```
study.optimize(
                func, <Objective function to optimize.
                int n_trials=None, <Number of trials to run.
                float timeout= None <Maximum time allowed to optimize study.
                )
```

*(Additional arguments available…)*

# Objectives

- Define an objective function that returns the metric(s) being maximized/minimized.
- You may use an individual objective function or define it as a method on a class which can be useful for handling additional parameters and configurations inside the objective function.
- The function should have a trial argument which passes a trial object that handles the current optimization attempt.

Ex: Loss,  accuracy, additional costs in time or resources.

# Ex: Defining our objective

```
def objective(trial : optuna.trial.Trial):

        number_of_layers = trial.suggest_int(name="n_layers", low = 1, high = 5)

        dropout = trial.suggest_float(name="dropout", low = 0.0, high = 0.5, log = True)

        learning_rate = trial.suggest_float(name="learning_rate", low = 1e-5, high= 1e-2, log = True)

        model = my_model(number_of_layers,dropout,learning_rate)

        result = model.train()

        return(result)
```

# Ex: Defining and running our study

```
Study=optuna.create_study(direction="minimize")

Study.optimize(objective, n_trials=50)
```

# Optuna dashboard

- Optuna-dashboard is a python package that allows you to host a GUI for optuna studies, allowing for manual management and inspection.
- To host optuna dashboard using sqlite storage you may open a terminal and run:

*optuna-dashboard --host address –port portnumber sqlite:///databasename.db*

For our hands on example we will host on the following address: http://127.0.0.1:8080

*optuna-dashboard --host 127.0.0.1 --port 8080 sqlite:///optuna_studies/optuna.db*

**The UI should now be accessible through your browser.**

# Hands on 1): Setting up an optimization run.

Checkout Tutorial-1.

Covers:

- Defining our objective.
- Making suggestions.
- Setting up study.
- Running the optimization task.

15 min.

# Pruning

To save on both time and computational resources, optuna supports termination of trials that does not show promise in the early stages of training.

Various pruning methods are available such as hyperband pruning, percentile pruning etc.

In our tutorial we will be using the median pruner, comparing the median of the metric across trials for a given timestep.

# Pruning

In between epochs or batches, we may report the metric we want to prune based on (for example validation loss at each epoch) using:

trial.report(current_value, epoch)

Then, the training loop can be terminated based on the condition that:

trial.should_prune()==True

# Hands on 2): Pruning

- Covers:
- Setting up a pruner.
- Checking pruning during training loop.
- Warmup and other options.

15 min.

# Experiment tracking (MLFlow)

MLFlow is a flexible platform allowing you to track and log your model in real-time locally or to shared servers, with various features supporting the evaluation, packaging, and deploying your models.

It has become a commonly used tool within the industry, but while it shines in a team setting it also makes working with large sets of models and experiments a lot more manageable for the individual as well.

# General structure

- Experiments: Separate different projects on the same server by logging them to different experiments. Can be identified by name string or int.
- Runs: Each training instance can be set to generate runs individually or as children to a parent run, useful with Optuna HPO. Each run has its own unique runID.
- Models: The direct machine learning product of your runs, models have an independent existence to their originating run.

# Setting up the MLFlow tracking server

- In a team setting you will usualy work with a shared mlflow server, but for this exercise we will be hosting a local server of our own.
- To set up the server for our exercise, open your terminalin the project root and activate the environment, then run:

```
mlflow server --backend-store-uri sqlite:///mlruns/mlflow.db --default-artifact-root file:./mlruns --host 127.0.0.1 --port 5000
```

This creates a server that stores artifacts and runs under /mlruns in the active directory.

# Connecting to mlflow in your code:

- Set the uri with: mlflow.set_tracking_uri(”uri”).
- Set the active experiment with mlflow.set_experiment(”experimentname”)

Ex:

mlflow.set_tracking_uri(http://127.0.0.1:5000)

mlflow.set_experiment(”my_experiment”)

# MLFlow logging

- Parameters: We may log parameters of our choice using:
  - mlflow.log_param(str key,  value) <For a single parameter.
  - mlflow.log_params(dict key,_value_dict) <For multiple parameters in a batch.
- Metrics: For metrics we may also want to include information such as which training step we are in:
  - mlflow.log_metric(str key,  value, int step)
  - mlflow.log_metrics(dict key_, value_dict)

# MLFlow logging

- Artifacts: The relevant products of our training. This can include everything from model parameters to figures generated by the evaluation step. These are stored according to the server's artifact storage, under the attached run.
  - mlflow.log_figure(fig, file_path) <Store a figure.
  - mlflow.log_text(fig, file_path) <Store a text.

# Active run

- If creating a run, then within that scope, logging functions without the runID specified will default to loggin under that run.

- EX, creating a run and setting the ”Hello” tag to ”World!:

```
mlflow.set_tracking_uri(http://127.0.0.1:5000)
mlflow.set_experiment(”my_experiment”)
with mlflow.start_run() as run:
        mlflow.set_tag(key="Hello", value="World!")
        model.train_and_log()
```

# Hands on 3: Logging to MLFlow

Covers:

- Connecting to MLFlow.
- Logging:
  - Tags.
  - Parameters.
  - Artifacts.

(15-20 mins)

# MLFlow models

- When logging a model in mlflow with mlflow.log_model a new model artifact is generated (under the runs models). These can be added globally to the model registry and served by mlflow models or in a container.
- When logging our models we can provide an input-output signature, package the source files and define the requirements, environment etc.

# Serving a registered model

With a model in the mlflow registry we may generate docker images for deployment with:

```
mlflow models build-docker –m ”models:/modelname/version” –n imagename
```

We may also host a local server using mlflow command line with:

```
mlflow models serve -m "models:/modelname/version" -p port
```

# Serving a registered model

NOTE! We need to set an environment variable pointing at our tracking server to connect the model server. Before using serve in , run the command:

(windows:) setx MLFLOW_TRACKING_URI <tracker host>:<tracer port>

(linux/mac:) export MLFLOW_TRACKING_URI= <tracker host>:<tracer port>

Ex: *setx MLFLOW_TRACKING_URI http://127.0.0.1:5000*

# Serving a registered model

In our exercise (running with default names and configurations) this will be:

```
mlflow models serve -m "models:/MLP_circles/1" --no-conda -p 5001
```

*(Where we disable the environment setup using –no-conda due to working in an environment already set up for the project.)*

# Hands on 4: Packaging models and serving.

- Covers:
- Logging the model.
- Trying out the model registry.
- Serving and calling your test model (?)

(15-20 mins)