OpusPocus

NMT Training Pipeline Manager

August 27

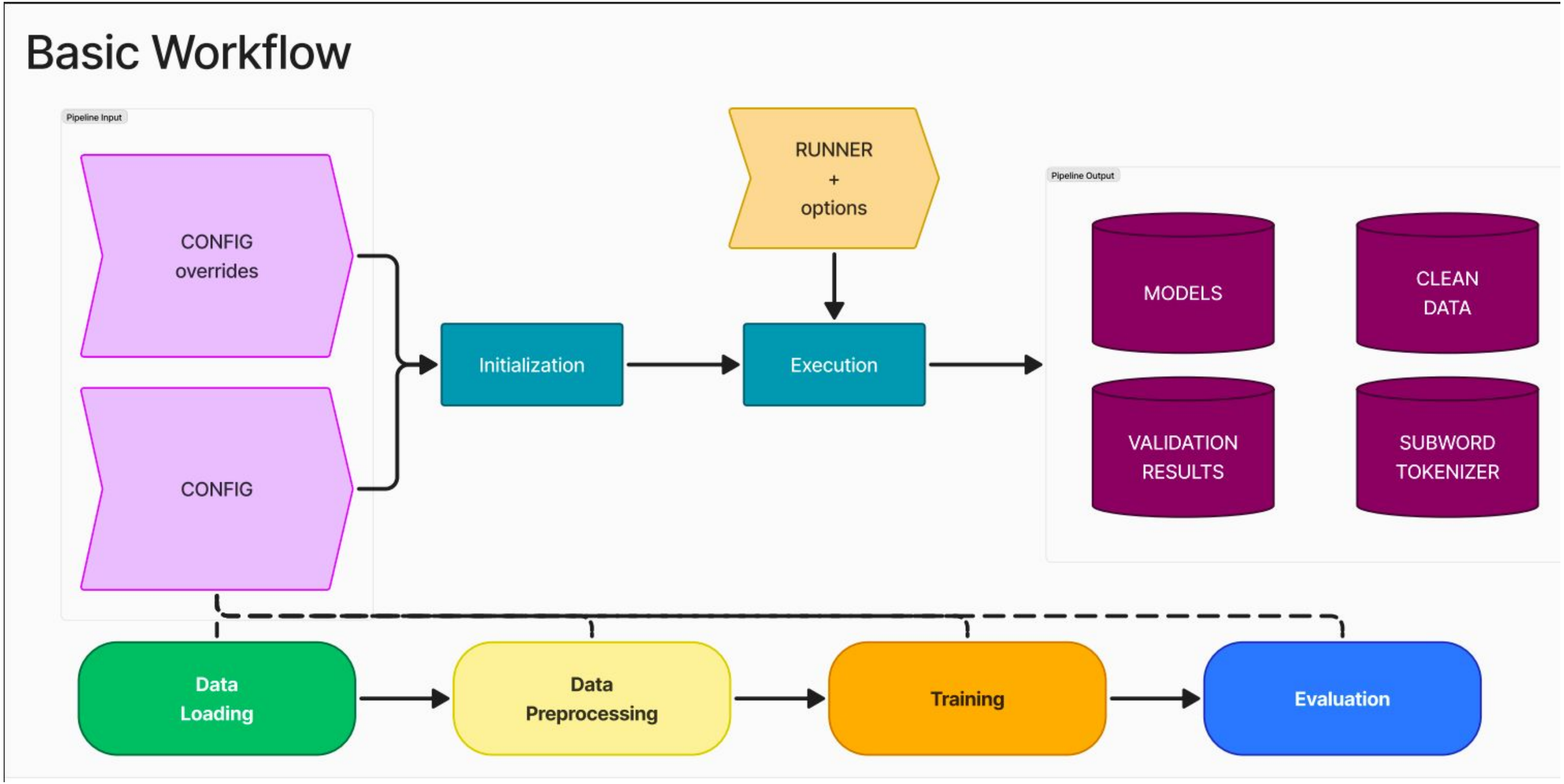
varis@ufal.mff.cuni.cz

Quick Introduction

Motivation

- NMT Training - too many stages to execute manually -> automation
- Existing solutions
    - Makefile, SnakeMake
        - too general,
        - hard to debug/maintain (when combined with HPC schedulers),
        - known issues on HPC (LUMI)
            - handling resubmission after time-limit
            - effective pipeline recovery after sharded task failure
- OpusPocus
    - focused only on NMT
        - in future could be extended to other NLP tasks
    - developed with MT community needs in mind
    - robust with regards to HPC (Slurm)

OpusPocus



Basic Workflow

OpusPocus – Use Cases

- Experiment replicability:
  - config describes full train pipeline (from data acquisition up to evaluation)
- Adaptation to many language (language pairs)
  - executing overwritten pipeline config
- Experiments with data preprocessing
  - Variations of the preprocessing pipeline + fixed model training pipeline
- Testing model variants
  - backtranslation
    - data mixing strategies
    - iterative backtranslation

Exercises

Installation

Python >= 3.9 (tested with Python 3.10.7)
(Use virtualenv or Conda to create a clean Python installation)

1. Clone the git repository (mtm2025 branch) and install dependencies:

```
$ git clone -b mtm2025 git@github.com:hplt-project/OpusPocus.git
$ cd OpusPocus
$ pip install -r requirements.txt
```

2. Download data:

```
$ mkdir data
$ scripts/get_data_hplt_v2.0.py --languages eu ca --data-dir data/
```

Installation (cont.)

Install Marian (ideally GPU version):

```
$ git clone git@github.com:marian-nmt/marian.git marian
$ mkdir marian/build && cd marian/build
$ cmake .. \
    -DCMAKE_BUILD_TYPE=Release \
    -DCOMPILE_CUDA=ON \
    -DCUDA_TOOLKIT_ROOT_DIR=<CUDA_ROOT_LOCATION> \
    -DUSE_CUDNN=ON \
    -DCUDNN_LIBRARY=<libcudnn.so_LOCATION> \
    -DCUDNN_INCLUDE_DIR=<CUDNN_INCLUDE_LOCATION> \
$ make -j 8
```

(At the end, the Marian should be located at OpusPocus/marian)

Basic Execution

```
$ ./go.py <subcommand> [options]
```

- Subcommands:
  - **run** - executes pipeline
  - **stop** - stop a running pipeline
  - **status** - print pipeline status
  - **traceback** - step dependency structure with their status

```
$ ./go.py <subcommand> --help
```

- lists subcommand options

Example 0: Running Minimal Pipeline

```
$ ./go.py run --pipeline-config config/00-pipeline.minimal.yml
```

1. No data processing
2. Single direction model training + evaluation

Check status:

```
$ ./go.py status --pipeline-dir experiments/en-eu/pipeline.minimal
```

Stop the pipeline:

```
$ ./go.py stop --pipeline-dir experiments/en-eu/pipeline.minimal
```

Continue execution:

```
$ ./go.py run --pipeline-dir experiments/en-eu/pipeline.minimal
--reinit-failed
```

Pipeline Config

- YAML format, processed with OmegaConf
  - easy to read, simple processing in Python
  - variable interpolation support
    - `${full.path.to.variable}` (e.g. `${pipeline.pipeline_dir}`)
    - or relative paths
      - `${.current_level_var}`
      - `${..level_above_var}`
      - `${...two_levels_above_var}`

Supports CLI overwriting:

```
$ ./go.py run --pipeline-config config/00-pipeline-minimal.yml
pipeline.src_lang=$lang pipeline.tgt_lang=en
```

Pipeline Config Structure

- pipeline:
  - **.steps** - list of steps and their dependencies
  - **.targets** - final pipeline steps to be executed
    - also implies execution of the target step dependencies
  - other variables - used as defaults for non-specified step-arguments

- runner:
  - **.runner** - name of the runner
  - **.runner_resources** - global resources for runner execution
  - other runner specific options

- step (pipeline.steps[i]):
  - *list* of step argument definitions
    - each step having a separate *dict* of arguments

Step Config Structure

- OpusPocusStep (general args):
  - **step** -
  - **step_label** -
  - **runner_resources** (dict) -
  - ***_step** - step-specific dependencies
    - represented by step_label of a given dependency
- CorpusStep derivatives:
  - take previous (corpus) step and apply a transformation to that corpus
  - args:
    - **src_lang** - (source) corpus language (required)
    - **tgt_lang** - target corpus language (optional)
    - **prev_corpus_step** - step containing corpus being transformed
- Default values:
  - taken from pipeline.<argument> name if available

Pipeline Directory

Created during **INITIALIZATION** (before execution)

- **pipeline_root/**
  - **pipeline.config** - configuration of the pipeline
  - **<step_label>/** - subdirectory containing the <step_label> step details

- **<step_label>/**:
  - **output/** - step-produced output
  - **logs/** - log files
  - **temp/** - temporary (input) files - deleted when step is **DONE**
  - **step.state** - current step execution state
  - **step.parameters** - step initialization parameters
  - **step.dependencies** - list of step dependencies
  - **step.command** - step executable
  - **runner.step_info** - information about a runner submission (job_id, ...)

Pipeline/Step States

Step States

- null (not created yet)
- **INITED** – intialized, step directory created
- **SUBMITTED** – submitted for execution by a RUNNER
- **RUNNING** – currently being executed
- **DONE** – execution 👍
- **FAILED** – execution 👎

PipelineStates similar + INIT_INCOMPLETE and PARTIALLY_DONE

Example 1: Bidirectional Translation Pipeline

```
$ ./go.py run --pipeline-config config/01-pipeline.bidirectional.yml
```

1. Add the opposite translation directions
2. Simplify config - move selected step arguments to global pipeline arguments

Global Pipeline variables

```
pipeline:
      src_lang: en
      tgt_lang: eu
      steps:
       -   step: raw
           step_label: test1
       -   step: raw
           step_label: test2
```

Equals to:

```
pipeline:
      steps:
       -   step: raw
           step_label: test1
           src_lang: en
           tgt_lang: eu
       -   step: raw
           step_label: test2
           src_lang: en
           tgt_lang: eu
```

Example 2: Execution on Slurm

```
$ ./go.py run --pipeline-config config/02-pipeline.bidirectional.slurm.yml
```

1. Change the default runner to “slurm” runner
2. Specify the available *runner_resources* for the runner

Runner Resources

- Runner-agnostic representation of the available execution resources
  - # gpus, # cpus, memory size, etc.
- Each runner implements conversion to its own representation
  - “runner_resources.gpus=4” ==(Slurm)=> “--gpus=4” option
- Each resource is also passed as OpusPocus environment variable to the executed script
  - “runner_resources.gpus” == $OPUSPOCUS_gpus

Example 3: Step-specific Execution Resources

```
$ ./go.py run --pipeline-config config/03-pipeline.bidirectional.resources.yml
```

1. We want to use different resources for different pipeline steps
    - step-specific *runner_resources* overwrite “global” *runner.runner_resources*

Example 4: Data Preprocessing

```
$ ./go.py run --pipeline-config config/04-pipeline.data_preprocess.yml
```

- preprocess input training data
  a. clean data with OpusCleaner
  b. decontaminate (remove training examples similar to valid/test data)
  c. “gather” corpora into a single corpus file

OpusCleaner - Data Filters

- must be located in the same directory as the corpus files
  - e.g: *data/my_corpus.en-eu.en.gz*
    => e.g.: *data/my_corpus.en-eu.filter.json*
  - no *.filter.json* file => cleaning is skipped
- example filters:
  - *config/opuscleaner_filters/*
  - copy the filters to your raw data directory (*data/en-eu/raw/v2*)

Can be created/edited:

- by directly editing the *.filter.json* files
- via OpusCleaner web UI
  - see https://github.com/hplt-project/OpusCleaner

Example 5: NMT with Backtranslation

```
$ ./go.py run --pipeline-config config/05-pipeline.backtranslation.yml
```

- train a mock model with BT data using the preprocessed data from the previous example
  - the previous pipeline must be in the **DONE** state
- “fake” monolingual data (using the parallel corpora instead)
- *auth* and *synth* data mixing via OpusTrainer

OpusTrainer - Configuration

- default config created by OpusPocus
  - limited expression
- more robust configuration directly via OpusTrainer config file:
  - see https://github.com/hplt-project/OpusTrainer
  - dataset scheduling, interleaving, curriculum learning, etc.
  - noise introducing schemes for better robustness
  - not dependent on MarianNMT
    - but tested mostly with MarianNMT

Future Work

- Multi-lingual training support
- Support for other training/decoding frameworks
  - HuggingFace
  - NLLB
  - LLM training (?)
- Integrating OpusDistillery for student model training
  - see https://github.com/Helsinki-NLP/OpusDistillery
- Fixing issues, improving user experience
  - check https://github.com/hplt-project/OpusPocus/issues