The YAML: an overview

João Lucas de Sousa Almeida, Daniela Szwarcman

April 13, 2025

The YAML format

If you are using the command-line interface (CLI) to run jobs using TerraTorch, so you must became familiar with YAML, the format used to configure all the workflow within the toolkit.

- Writing a YAML file is very similar to coding, because even if you are not directly handling the classes and others structures defined inside a codebase, you need to know how they work, their input arguments and their position in the pipeline.
- In this way, we could call it a "low-code" task.
- The YAML file used for TerraTorch has an almost closed format, since there are a few fixed fields that must be filled with limited sets of classes, which makes easier for new users to get a pre-existing YAML file and adapt it to their own purposes.
- In the next sections, we describe each field of a YAML file used for Earth Observation Foundation Models (EOFM) and try to make it clearer for a new user. However, we will not go into detail, since the complementary documentation (Lightning, PyTorch, ...) must fill this gap.

Trainer: global arguments

In the section called **trainer** are defined the arguments that must be sent to the Lightning Trainer object. If you need a deeper explanation about this, check the Lightning's documentation. In the first lines we have:

- **accelerator**: refers to the kind of device is being used to run the experiment. Usually **cpu** and **gpu**, but if you set **auto**, it will automatically select allocate the GPU is that is available or otherwise run on CPU.
- **strategy**: is related to the kind of parallelism is available. As we have usually ran the experiments using a single device for fine-tuning or inference, we do not care about it and choose the option **auto** by default.
- **devices**: indicates the list of available devices to use for the experiment. Leave it as **auto** if you are running with a single device.
- **num_nodes**: is self-explanatory. We have mostly tested TerraTorch for single-node jobs, so, it is better to set it as 1 for now.
- **precision**: is the kind of precision used for your model. **16-mixed** have been an usual choice.

Trainer: logging

Just below this initial stage, we have **logger**:

```
logger:
    class_path: TensorBoardLogger
    init_args:
        save_dir: tests/
        name: all\_ecos_random

```

In this field we define the configuration for logging the model state. In this example we are using Tensorboard, and saving all the logs in a directory *tests/all_ecos_random*. Others frameworks, as MLFlow are also supported. Check the Lightning documentation about logging for a more complete description.

Trainer: callbacks

The **callbacks** field:

```
callbacks:
  - class_path: RichProgressBar
  - class_path: LearningRateMonitor
    init_args:
      logging_interval: epoch
  - class_path: EarlyStopping
    init_args:
      monitor: val/loss
      patience: 100
```

Represents a list of operations that can be invoked with determined frequency. The user is free to add others operations from Lightning or custom ones. In the current config we are basically defining: a progress bar to be printed during the model training/validation and a learning rate monitor, determined to call early-stopping when the model shows signals of overfitting.

Trainer: final part

The rest of the arguments are:

```
16  max_epochs: 1
17  check_val_every_n_epoch: 1
18  log_every_n_steps: 20
19  enable_checkpointing: true
20  default_root_dir: tests/
```

- **max_epochs**: the maximum number of epochs to train the model. Notice that, if you are using early-stopping, maybe the training will finish before achieving this number.
- **check_val_every_n_epoch**: the frequency to evaluate the model using the validation dataset. The validation is important to verify if the model is tending to overfit and can be used, for example, to define when update the learning rate, or to invoke the early-stopping.
- **enable_checkpointing**: it enables the checkpointing, the action of periodically saving the state of the model to a file.
- **default_root_dir**: the directory used to save the model checkpoints.

Datamodule

In this section, we start directly handling TerraTorch's built-in structures. The field **data** is expected to receive a generic datamodule or any other datamodule compatible with Lightning Datamodules, as those defined in our collection of datamodules.
In the beginning of the field we have:

```
21 data:
22   class_path: GenericNonGeoPixelwiseRegressionDataModule
23   init_args:
```

It means that we have chosen the generic regression datamodule and we will pass all its required arguments below **init_args** and with one new level of indentation. The best practice here is to check the documentation of the datamodule class you are using (in our case, here) and verify all the arguments it expects to receive ant then to fill the lines with <**argument_name**>: <**argument_value**>. As the TerraTorch and Lightning modules were already imported in the CLI script (**terratorch/cli_tools.py**), you do not need to provide the complete paths for them. Otherwise, if you are using a datamodule defined in an external package, indicate the path to import the model, as **package.datamodules.SomeDatamodule**.

Model

The field **model** is, in fact, the configuration for **task** + **model**:

```
model:
  class_path: terratorch.tasks.PixelwiseRegressionTask
  init_args:
    model_args:
      decoder: UperNetDecoder
      pretrained: false
      backbone: prithvi_eo_v2_600
      backbone_drop_path_rate: 0.3
      backbone_window_size: 8
      decoder_channels: 64
      num_frames: 1
      in_channels: 6
      bands:
        - BLUE
        - GREEN
        - RED
        - NIR_NARROW
        - SWIR_1
        - SWIR_2
```

Model: continuation

Notice that there is a field **model_args**, which it is intended to receive all the necessary configuration to instantiate the model itself, that means, the structure **backbone + decoder + head**. Inside **model_args**, it is possible do define which arguments will be sent to each component by including a prefix to the argument names, as **backbone_<argument>** or **decoder_<other_argument>**. Alternatively, it is possible to pass the arguments using dictionaries **backbone_kwargs**, **decoder_kwargs** and **head_kwargs**.

```
43   model_args:
44      decoder: UperNetDecoder
45      pretrained: false
46      backbone: prithvi_eo_v2_600
47      backbone_kwargs:
48        drop_path_rate: 0.3
49        window_size: 8
```

The same recommendation made for the **data** field is repeated here, check the documentation of the task and model classes backbones, decoders and heads you are using in order to define which arguments to write for each subfield of **model**.

Model: final part

The last fields are related to the others main parameters (as loss configuration):

```
    loss: rmse
    ignore_index: -1
    freeze_backbone: true
    freeze_decoder: false
    model_factory: PrithviModelFactory
```

And the tiled inference. A kind of inference in which the model is applied to small pieces of the image instead of the full one.

```
    tiled_inference_parameters:
       h_crop: 224
       h_stride: 192
       w_crop: 224
       w_stride: 192
       average_patches: true
```

Optimization and Learning Rate Scheduler

The last two fields of out example are the configuration of the optimizer and the lr scheduler. Those fields are mostly self-explanatory for users already familiar with machine learning:

```
optimizer:
  class_path: torch.optim.AdamW
  init_args:
    lr: 0.00013524680528283027
    weight_decay: 0.047782217873995426
lr_scheduler:
  class_path: ReduceLROnPlateau
  init_args:
    monitor: val/loss
```

Check the PyTorch documentation about optimization to understand them more deeply.