NIMS Federated Learning

Release 2.2

NIMS

Nov 14, 2024

NOTES

1 Getting Started **1**
- 1.1 Content of the deliverable . . . 1
- 1.2 Installation . . . 2
- 1.3 Command line . . . 2

2 Configuration Overview **5**
- 2.1 Model Configuration . . . 5
- 2.2 Mila Configuration (Federated learning) . . . 5

3 Framework Overview **7**
- 3.1 Project Structure . . . 7
- 3.2 Abstractions . . . 9
- 3.3 Dependency Injection . . . 9
- 3.4 Dynamic Configuration . . . 9
- 3.5 Events & Observers . . . 11
- 3.6 Models . . . 11
- 3.7 Configuration . . . 11

4 Using SSL (secure) connection **13**
- 4.1 On the server system . . . 13
- 4.2 On the client system . . . 14
- 4.3 Configuration set up: . . . 14

5 Rerun experiments using provided config **17**
- 5.1 Cross validation experiments . . . 17

6 Use Bayesian optimization **19**
- 6.1 Set up config for Bayesian optimization . . . 19

7 Model Configuration **21**
- 7.1 model . . . 21
- 7.2 loader . . . 21
- 7.3 featurizer . . . 21
- 7.4 transformer . . . 22
- 7.5 splitter . . . 22
- 7.6 criterion . . . 22
- 7.7 optimizer . . . 22
- 7.8 scheduler . . . 23
- 7.9 is_stepwise_scheduler (default: `True`) . . . 23
- 7.10 is_finetuning (default: `False`) . . . 23
- 7.11 output_path . . . 23

7.12 checkpoint_path (default: `None`) . . . 23
7.13 threshold (default: `0.5`) . . . 23
7.14 cross_validation_folds (default: `5`) . . . 23
7.15 train_split (default: `train`) . . . 24
7.16 test_split (default: `test`) . . . 24
7.17 train_metrics (default: `[]`) . . . 24
7.18 test_metrics (default: `[]`) . . . 24
7.19 epochs (default: `200`) . . . 24
7.20 batch_size (default: `32`) . . . 24
7.21 use_cuda (default: `true`) . . . 24
7.22 cache_location (default: `/tmp/federated/`) . . . 24
7.23 clear_cache (default: `False`) . . . 24
7.24 log_frequency (default: `20`) . . . 25
7.25 log_level (default: `info`) . . . 25
7.26 log_format (default: `""`) . . . 25
7.27 target_metric (default: `roc_auc`) . . . 25
7.28 optuna_trials (default: `1000`) . . . 25
7.29 subset (default: `None`) . . . 25
7.30 observers (default: `{}`) . . . 25

8 Federated Learning (Mila) 27
8.1 Server Configurations . . . 28
8.2 Client Configurations . . . 31
8.3 Mean CV Configuration . . . 32

9 Aggregators 33
9.1 Plain aggregation . . . 33
9.2 Weighted aggregation . . . 34
9.3 Benchmark aggregation . . . 34
9.4 Fedrox aggregation . . . 34

10 Criterions 37

11 Featurizers 39

12 Loaders 43

13 Models 45
13.1 Graph Convolutional Networks . . . 45
13.2 Message Passing Neural Networks . . . 47
13.3 Triplet Message Passing Networks . . . 47
13.4 Linear Networks . . . 48
13.5 SimpleMLPNetwork . . . 48
13.6 Convolutional Networks . . . 48
13.7 Protein Ligand . . . 49

14 Splitters 51
14.1 Shared argument . . . 51
14.2 Stratified . . . 52
14.3 Categorical . . . 52

15 Transformers 55

CHAPTER

ONE

GETTING STARTED

Contents

- *Getting Started*
  - *Content of the deliverable*
  - *Installation*
  - *Command line*
    - *nims-federated-learning-base*
      - `train`
      - `eval`
      - `mean_cv`
      - `predict`
      - `optimize`
      - `find_best_checkpoint`
    - *nims-federated-learning*
      - `server`
      - `client`
      - `mean_cv`

1.1 Content of the deliverable

In the deliverable we provided there is the various file and folder describe below:

- **federated-learning (folder)**
  Contain the code for the project.
- **docs (folder)**
  Contain a pdf and HTML version of this documentation.
- **experiment_configs (folder)**
  Contain all experiment configs necessary to rerun our experiments.
- **README.md (file)**
  Contains a subset of the documentation

1.2 Installation

Use the provided conda snapshot:

```
conda env create -f environment.yml  # first time only
conda activate federated
bash install_additional_dependencies.sh  # first time only
```

1.3 Command line

There are two commands line input possible:

- `nims-federated-learning-base`
- `nims-federated-learning`

1.3.1 nims-federated-learning-base

This command is used for standard training (no federated-learning) and optimization there is a variety of command possible.

It expects a couple of argument:

```
nims-federated-learning-base {job} {config}
```

Where `job` is one of the task detail below and `config` the path to a *model configuration*.

`train`

Run a standard training base on the configuration provided.

`eval`

Launch evaluation of the model and return the target metric for the *checkpoint_path* provided.

`mean_cv`

Run cross validation for the given config and return the mean average of the target metric.

`predict`

Run inference on the dataset *test_split* and save result in a *predictions.csv* file in the *output_path* directory.

`optimize`

Run optimization with optuna base on the configuration provided

more detail on optimisation runs

`find_best_checkpoint`

Find the best checkpoint base on the *output_path* and *epochs* parameters. It will evaluate all checkpoint in the directory until provided *epochs* number and return the best one as well as it's target metrics.

1.3.2 nims-federated-learning

This command is used for federated-learning training.

It expects a couple of argument:

```
nims-federated-learning {job} {config}
```

Where `job` is one of the three tasks `server`, `client` or `mean_cv` and `config` the path to the expected config format to the task, *see the expected format here*.

`server`

Launch a server node (should always be the first node to be launch)

`client`

Launch a client node

`mean_cv`

Wrapper around multiple launch of both server and clients. It will launch *num_folds* number of experiment and save all results and prediction to the *output_path*.

This is a scenario only useful for testing different model and should be run on only one computer since one program manage all client and server. This is due to the fact that when splitting the dataset, we must have the full dataset at hand to make the different folds. And all clients must split the dataset in the exact same way.

CHAPTER

TWO

CONFIGURATION OVERVIEW

There are two mains sets of configuration. One is for the use of `nims-federated-learning-base` command line argument, whereas for `nims-federated-learning` both type are needed.

2.1 Model Configuration

Configuration used to make a standard training. So it provides information about the splitter, the optimizer, etc.

more details

2.2 Mila Configuration (Federated learning)

There is three set of configuration.

- **Server configuration:**
  Used to start a server node
- **Client configuration:**
  Used to start a client node
- **Mean_cv configuration:**
  Used in parallel with Server and Client config to launch a cross validation experiment.

  This can only be used on one machine where the dataset is being split across clients node and the folds are managed between each run.

more details

CHAPTER

THREE

FRAMEWORK OVERVIEW

3.1 Project Structure

The code is divided into several folders:

- `data`: contains datasets, checkpoints, certificates, logs, and everything else data related. This folder is not fixed. Data can be stored anywhere else on the disk.
- `lib`: contains all the custom code implemented for data preprocessing, training, inference, and experimenting with models in general.
- `mila`: contains the communication module (federated learning)
- `vendor`: contains chunks of code from 3rd party libraries
- `run.py`: is the entry point for experiments (models, not federated learning)
- `environment.yml`: are installation environments and scripts

The main codebase `lib` is furthermore split into 4 folders:

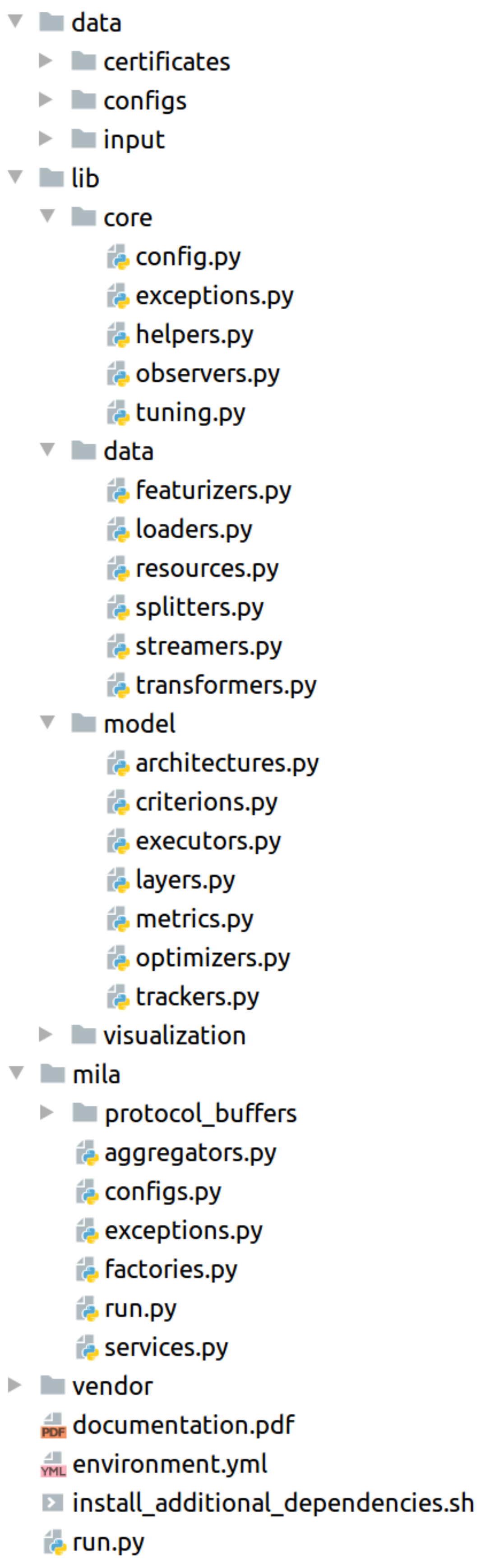
- `core`: contains often used functionality and helper functions
- `data`: contains data handles and pre-processing tools
- `model`: contains architectures, metrics, and modeling tools
- `visualization`: contains functionality from visualizing results

The `core` folder contains:

- `config.py`: for configuration parsing
- `exceptions.py`: for exception handling
- `helpers.py`: for functionality like caching, time tracking, reflection, and dependency injection
- `observers.py`: for event management
- `tuning.py`: for Bayesian optimization

The `data` folder contains:

- `featurizers.py`: for preparing input fields
- `loaders.py`: for loading raw data formats from disk
- `resources.py`: for common structures
- `splitters.py`: for data splitting
- `streamers.py`: where we combine data loading and preprocessing tools
- `transformers.py`: for preparing output fields



```
▼ data
    ▶ certificates
    ▶ configs
    ▶ input
▼ lib
    ▼ core
        config.py
        exceptions.py
        helpers.py
        observers.py
        tuning.py
    ▼ data
        featurizers.py
        loaders.py
        resources.py
        splitters.py
        streamers.py
        transformers.py
    ▼ model
        architectures.py
        criterions.py
        executors.py
        layers.py
        metrics.py
        optimizers.py
        trackers.py
    ▶ visualization
▼ mila
    ▶ protocol_buffers
    aggregators.py
    configs.py
    exceptions.py
    factories.py
    run.py
    services.py
▶ vendor
documentation.pdf
environment.yml
install_additional_dependencies.sh
run.py
```

The `mila` folder contains:

- `protocol_buffers`: for service contracts and gRPC messages
- `aggregators.py`: for aggregator implementations
- `configs.py`: for server and client configuration parsing
- `exceptions.py`: for exception handling
- `factories.py`: for abstractions
- `run.py`: is the entry point for federated learning (both server and clients)
- `services.py`: implements the communication between server and clients

The vision and efforts for the provided solution are centered are 3 key points:

- Good quality and clean code
- Extensible with minimum effort
- Highly configurable and easy to use

Several concepts and design patterns are utilized as a means of improving these characteristics. Some of these major components are briefly discussed in the following sections.

3.2 Abstractions

Most of our components, like loaders, featurizers, networks, or execution pipelines are designed by contract. For each group of components, we first design an abstract class (ie: AbstractLoader) with common functionality, but more importantly, common methods. The abstract classes are then extended to implement detailed solutions (ie: CsvLoader).

Then, if high level components (like the streamers) rely on low level ones (like the loaders), it is enough for us to specify the abstraction, and the high level module will be able to handle all subtypes of it (it acts as a contract). In object-oriented design, this is referred to as dependency inversion.

3.3 Dependency Injection

Dependency injection makes a class independent of its dependencies (ie: input arguments). This is achieved by decoupling the use of a component from its instantiation.

Among other things, this means the code of a lower level objects can be changed without having to modify higher level objects which depend on it. This can be a very powerful concept, because it allows us to easily add new functionality without having to write a lot of additional code.

For a more visual way to see the benefits of this concept, see the next section.

3.4 Dynamic Configuration

To run experiments, we require a configuration file in JSON format (details in a later section). While some of the structure is rigid, for the most part, configurable options are directly linked to the argument lists of the components' constructor. This is achieved with *Dependency Injection*.

For example, let us assume we have a `Calculator` object we want to make use of:

```python
# file: custom/helpers.py

class Calculator:
```

(continues on next page)

(continued from previous page)

```python
    def __init__(self, x: int, y: int):
        self._x = x
        self._y = y

    def sum(self) -> int:
        return self._x + self._y
```

To make use of this object, no changes have to be made to the configuration parsing logic whatsoever. We just point to the class we want to use and specify its arguments:

To make use of this object, no changes have to be made to the configuration parsing logic whatsoever. We just point to the class we want to use and specify its arguments:

```json
{
    "helper": {
        "type": custom.helpers.Calculator,
        "x": 2,
        "y": 3
    }
}
```

Input arguments do not have to be numeric though, we can use any objects whatsoever. If additional classes are required, they will be instantiated recursively. As an example, this would also work:

```python
# file: custom/computers.py

from custom.helpers import Calculator

class Computer:

    def __init__(self, calculator: Calculator, constants: list):
        self._calculator = calculator
        self._constants = constants

    def sum(self, x: int, y: int) -> int:
        return self._calculator.sum(x, y)
```

```json
{
    "computer": {
        "type": "custom.computers.Computer",
        "calculator": {
            "type": custom.helpers.Calculator,
            "x": 2,
            "y": 3
        },
        "constants": [1, 5]
    }
}
```

We make use of this functionality a lot, and it makes it very easy to use already implemented features like loss functions from the core Pytorch library or graph convolutional operators from Pytorch Geometric. More importantly, we can make use of them without having to write large chunks of complicated logic, or any code at all for most cases.

On the negative side, the compatibility can make the configuration less straight forward and require a level of knowledge about the underlying arguments. We make this easier by providing plenty of examples on how to configure experiments (see the `experiment_configs` folder).

3.5 Events & Observers

Not everything can be solved with dependency injection. Some areas of any framework are just inherently complicated, complicated code is never good because it is messy, hard to maintain and more error-prone.

One good example is the training pipeline. Even for minimal functionality, we need to load checkpoints, initiate data loaders, perform forward and backward passes, log progress, and compute metrics. However, as a central area where many components connect, usually things do not stop there and extra functionality continue to pile up in the area.

To somewhat mitigate this problem, we introduced an event manager, also called an observer pattern. In an observer pattern, we keep track of a list of dependents which are automatically notified when important events happen. In case of the training pipeline for example, we can define such a dependent to receive a payload containing the model and optimizer right before backpropagation happens. Of course, we can then add any changes we want using that payload.

There are several benefits to this approach.

- We will not have an ever-growing training script for one
- Due to dynamic configuration, anyone can add their own event handlers without having to touch the core codebase at all
- This is very useful for backwards compatibility as well, because changes will not be overwritten upon update

However, adding too many observers which modify the original behavior can hide or cause dependency issues, and make the functionality harder to debug.

3.6 Models

Models live under the `lib/` folder.

Each experiment requires a configuration file to run, which we describe in the next section. Furthermore, for the models to be compatible with Mila, it has to extend 2 abstract classes from the Mila library.

- The `AbstractConfiguration` class - responsible for configuration handling
- The `AbstractExecutor` class - responsible for starting the train, validation, and inference processes.

3.7 Configuration

Configuration files are stored in JSON format and parsed by a class extending `AbstractConfiguration`. When using Mila with a `BenchmarkedAggregator`, the extended configuration must also include a `checkpoint_path` argument.

All experiments and operations make use of the same configuration file. Certain fields are dynamic and used for dependency injection. In these cases a “type” subfield will specify the desired class type, while all other subfields will be considered arguments for that class. Class arguments themselves can contain a “type” subfield, in which case additional objects will be instantiated recursively.

CHAPTER

FOUR

USING SSL (SECURE) CONNECTION

In order to have secure connection we are using SSL certificate to make the connection between client and server. Here we will detail the step to create those certificates.

4.1 On the server system

We first need to create the root private key and a self-signed root certificate.

We generate an RSA private key with des3 encryption method with the name `rootCA.key` with a size of 2048 bits.

```
openssl genrsa -des3 -out rootCA.key 2048
```

We then generate a certificate requests (CSRs) certificate name `rootCA.pem` based on the rootCA.key generated valid for 1024 days.

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
```

Next, we create a server private key, a server certificate signing request (CSR), and we sign the CSR using the root certificate.

We generate server key:

```
openssl genrsa -out server.key 2048
```

Generate a server certificate signing request (CSR):

```
openssl req -new -key server.key -out server.csr
```

Note

Here you must indicate *Common Name* that will be use to connect to this server This name will be use in the client configuration, the default value is :

`"options":  "[[grpc.ssl_target_name_override", "localhost"]]`

Output example

```
Country Name (2 letter code) [AU]:.
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:.
Organization Name (eg, company) [Internet Widgits Pty Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []:myserver (use localhost for default)
Email Address []:
```

Finally sign the certificate using the rootCA.pem and rootCA.key previously generated.

```
openssl x509 -req -in server.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out␣
↪server.crt -days 500 -sha256
```

4.2 On the client system

The client creates a private key and a certificate signing request (CSR).

```
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr
```

Note

The client will need to enter a Common Name which needs to be the same has the one in the above generated CSR.

Client send its CSR to the server which signs it and send it back along with *rootCA.pem* to the client. So on the server system the following command needs to be run.

```
openssl x509 -req -in client.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out␣
↪client.crt -days 500 -sha256
```

4.3 Configuration set up:

Then we can launch a training by modifying the following parameters:

- **Server side**:

```
{
    "use_secure_connection": True,
    "ssl_private_key": "server.key",
    "ssl_cert": "server.crt",
    "ssl_root_cert": "rootCA.pem"
}
```

`use_secure_connection (default=false)`

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

`ssl_private_key (default="data/certificates/server.key")`

gRPC secure communication private key

`ssl_cert (default="data/certificates/server.crt")`

gRPC secure communication SSL certificate

`ssl_root_cert (default="data/certificates/rootCA.pem")`

gRPC secure communication trusted root certificate

- **Client side**:

```
{
    "use_secure_connection": True,
    "ssl_private_key": "client.key",
    "ssl_cert": "client.crt",
    "ssl_root_cert": "rootCA.pem"
}
```

`use_secure_connection (default=false)`

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

`ssl_private_key (default="data/certificates/client.key")`

gRPC secure communication private key

`ssl_cert (default="data/certificates/client.crt")`

gRPC secure communication SSL certificate

`ssl_root_cert (default="data/certificates/rootCA.pem")`

gRPC secure communication trusted root certificate

RERUN EXPERIMENTS USING PROVIDED CONFIG

In the delivery package, there is *experiment_configs* folder containing all config used for training the experiments.

Note

There might be some path mismatch in the config between your environment and the one we used for training. We tried to fix those in the *experiment_configs* but the config present in *result* are still using our path system.

5.1 Cross validation experiments

Most experiment are federated-learning cross validation experiment.

To run those use you need to update the file *experiment_configs/mila_config/mean_cv.json* and indicate which experiment you would like to rerun. For all our experiment the *seed* parameter was 42 and the *num_folds* was 10. You only need to modify *cfg_dir* and provide a path to a directory inside *mila_config* as well as *output_path* to indicate where results will be stored.

Note

Experiment base on material splitting are in the categorical folder since they are using the categorical splitter.

For example if you provide the following config:

```json
{
    "cfg_dir": "experiment_configs/mila_config/base/epochs/2",
    "seed": 42,
    "num_folds": 10,
    "output_path": "result_dir_path",
    "log_level": "DEBUG",
    "save_results": false
}
```

And run:

```
nims-federated-learning mean_cv experiment_configs/mila_config/mean_cv.json
```

It will launch the server based on the config inside the directory and launch both client as well. It will repeat the process 10 times for each fold. So there will be a total of 10 servers and 20 clients created. The 10 runs does not run in parallel.

At the end the mean result of each run will be saved inside a pickle file called `result_pickle.pkl`. You can load it to review the result.

An example of result format, each element in the list correspond to the result of one fold:

```
{'log10timetorupture[h]':
[
    0.22755610241596613,
    0.276156495748355,
    0.2662610935919585,
    ...
    0.25868318230217774
]
}
```

If *save_results* is set to true, the best checkpoints of each fold training as well as the test set inference of all folds will be generated in *output_path*. For an example you can look into the *results* folder which contains this type of results for the material splitting experiments.

Note

For categorical experiment if you want to rerun some experiment without mean_cv set up, you might have to change the configuration. *See splitters information for more details <../modules_details/splitters>*

CHAPTER

SIX

USE BAYESIAN OPTIMIZATION

Optuna is an open source hyperparameter optimization framework we are using for the Bayesian optimization.

In order to use Optuna, you need to update your *model configuration <../configuration_details/configuration_model>* in a certain way to enable the parsing of the argument to optimize.

Once you finished updated your config, simply run:

```
nims-federated-learning-base optimize path_to/config.json
```

6.1 Set up config for Bayesian optimization

We have a parser to recognize which fields are supposed to be optimized with Optuna. You need to follow the following rules:

- each placeholder should have a name and some options, separated by an equal sign - ie: `"{{{name=options}}}"`
- options separated by `|` will be categorical - ie: `"{{{aggregate=mean|sum|max}}}"`
- numeric values should have 3 options separated by a dash `-`. ie: `"{{{dropout=min|max|step}}}"`
- **The first value is the minimum value**
    - The second value is the maximum value
    - The third value is the incremental step between the minimum and the maximum
    - The [minimum, maximum] is a closed interval
- if numeric values contain a dot ".", a float value will be suggested ie: `"{{{dropout=0.0-0.7-0.1}}}"`
- if numeric values do not contain a dot ".", an int value will be suggested ie: `"{{{layers=2-5-1}}}"`

So for example if I want to test various values of dropout in my model I will change the model parameter with:

```
"model": {
    "type": "SimpleMLP",
    "in_features": 22,
    "out_features": 1,
    "hidden_features": 512,
    "n_layer": 5,
    "dropout": "{{{dropout=0.0-0.7-0.1}}}"
}
```

At the end of the study the result of the best parameters will be displayed with its target metric results.

CHAPTER

SEVEN

MODEL CONFIGURATION

7.1 model

The model option is used to specify which model architecture to use and the options of that model. It is a dynamic field and used to instantiate torch modules which extend `lib.model.architectures.AbstractNetwork`.

As a dynamic field, the "type" argument is used to specify which class to use. At the moment, we support 7 architectures as input for the "type" option:

- `graph_convolutional`
- `message_passing`
- `triplet_message_passing`
- `linear`
- `convolutional`
- `protein_ligand`
- `simple_mlp`

more details

7.2 loader

The loader option is used to configure how raw data files are loaded. It is a dynamic field used to instantiate components which extend `lib.data.loaders.AbstractLoader`.

We support 3 loaders at this time, which are specified in the `type`:

- **csv**: For comma-separated values
- **excel**: For Excel spreadsheets (only one spreadsheet can be loaded at a time)
- **sdf**: For SDF files (where the whole dataset is stored in a single SDF file)

more details

7.3 featurizer

Featurizers prepare the input features for the training or inference. The expected input is a list, and therefore we can specify more than one featurizer per experiment. Featurizers are dynamic fields used to instantiate components extending `lib.data.featurizers.AbstractFeaturizer`.

As any dynamic field, the featurizer to be used is specified with the `type` option.

more details

7.4 transformer

Transformers are very similar to featurizers, expect they are applied to output features, and are generally used to normalize values. Transformers are dynamic fields used to instantiate components extending `lib.data.transformers.AbstractTransformer`.

Each transformer implements an "apply" and a "reverse" operation, which converts the values back after inference (for prediction only, does not apply to metrics). Similar to featurizers, multiple transformers can be used in the same experiment

more details

7.5 splitter

The last of the major abstraction based dynamic settings, the "splitter" specifies how to split the dataset. These options are used to instantiate components extending `lib.data.splitters.AbstractSplitter`.

more details

7.6 criterion

The criterion option is used to specify the loss function. It is a dynamic option, however, it is not relying on abstractions.

Users can specify any native Torch or custom written loss functions.

more details

7.7 optimizer

Is a dynamic option used to specify and configure the optimizer. As any other dynamic option, the class is specified with the "type" argument, and everything else is injected into the constructor:

```
"optimizer": {
    "type": "torch.optim.Adam",
    "lr": 0.01,
    "weight_decay": 0.00056
},
```

For a list of native PyTorch optimizers, please see the pytorch documentation on optim. Of course, custom optimizers can be used as well. As an example, we provide support for the recent AdaBelief optimizer:

```
"optimizer": {
    "type": "lib.model.optimizers.AdaBelief",
    "weight_decay": 0,
    "betas": [0.9, 0.999]
},
```

7.8 scheduler

Schedulers are used to adjust the learning rate during training. This is a dynamic option very similar to the criterion or the optimizer.

For a list of native Pytorch learning rate schedulers, please visit the pytorch documentation.

```
"scheduler": {
    "type": "torch.optim.lr_scheduler.OneCycleLR",
    "max_lr": 0.01,
    "epochs": 200,
    "pct_start": 0.3,
    "div_factor": 25,
    "final_div_factor": 1000
},
```

7.9 is_stepwise_scheduler (default: `True`)

A static field. If true, we perform scheduler updates after every forward pass. Otherwise, we perform the update after every epoch.

7.10 is_finetuning (default: `False`)

A static field. This should remain `False` if one wishes to continue training after a reboot, and be set to `True` when we wish to train on a new dataset using a previous checkpoint. When this option is `True`, only model weights will be loaded, otherwise we also load the optimizer, scheduler, and epoch tracking information.

7.11 output_path

Points to the location where checkpoints will be saved to.

7.12 checkpoint_path (default: `None`)

A static field marking the path to a checkpoint. Some operations like single checkpoint evaluation or inference require a fixed checkpoint. One can also specify a checkpoint path if they wish to continue training on a previous task, or for fine-tuning on another dataset.

7.13 threshold (default: `0.5`)

Specifies what threshold to use when converting logits to predictions. This is going to affect certain metrics (like accuracy) and predicted values (when running inference).

Note: The threshold is used only for classification tasks.

7.14 cross_validation_folds (default: `5`)

How many folds to split the data into when performing cross-validation?

7.15 train_split (default: `train`)

Specify which split to use for training. This should be one of the keys specified in the `splits` option of the `splitter`.

7.16 test_split (default: `test`)

Specify which split to use for evaluation and inference. This should be one of the keys specified in the `splits` option of the `splitter`.

7.17 train_metrics (default: `[]`)

Specify which metrics to report during training. This option is expected to be a list of strings, thus multiple metrics can be specified. But please note that metric computations can slow down the training process.

List of metrics:

- **Regression**
  mae, mse, rmse, r2, pearson, spearman, kl_div, js_div,chebyshev, manhattan, rank_quality
- **Classification**
  roc_auc, pr_auc, accuracy, precision, recall, f1, cohen_kappa, jaccard

7.18 test_metrics (default: `[]`)

Specify which metrics to report during evaluation. This option is expected to be a list of strings, thus multiple metrics can be specified.

7.19 epochs (default: `200`)

The number of training iteration upon seeing all datapoints

7.20 batch_size (default: `32`)

The number of datapoints to use (in all operations) in a single iteration.

7.21 use_cuda (default: `true`)

If true, and if an NVIDIA graphics card is available, the model will use GPU acceleration.

7.22 cache_location (default: `/tmp/federated/`)

A folder where cached objects will be stored in. It is recommended to monitor the size of this directory.

7.23 clear_cache (default: `False`)

When set to `True`, cached data for the specific experiment will be flushed.

7.24 log_frequency (default: `20`)

The number of iterations an update will be printed to the console when training the model.

7.25 log_level (default: `info`)

How many logs should be printed. The available options are: `debug`, `info`, `warn`, `error`, and `critical`.

7.26 log_format (default: `""`)

Can be used to include additional details for logged messages, like timestamps.

7.27 target_metric (default: `roc_auc`)

Specify which metric to use as feedback for Bayesian optimization. This option is also used when to find best performing checkpoints.

See *train_metrics* for a list of metrics

7.28 optuna_trials (default: `1000`)

Number of trials that should be performed when running Bayesian Optimization.

7.29 subset (default: `None`)

The subset is a static, but composite setting. It expects a dictionary containing an `id` and a `distribution` field.

The option is used to run experiments on a subset of the whole dataset, and is very helpful for distributed learning experiments. Otherwise, it provides little use, as the specific subsets can be obtained using the regular splitter functionality.

The `distribution` setting specifies a list of fractions the dataset should be split into, and the expected value is a list of floating point values which sum up to 1. The `id` setting specifies which subset should be used from the list of "distribution"s and is a 0-based index.

As an example, the below setting will split the dataset into 2 equal proportions, and will use the first half of the split for training.

```
"subset": {
    "id": 0,
    "distribution": [0.5, 0.5]
}
```

7.30 observers (default: `{}`)

Observers are handlers which listen for and handle incoming events. We set up various event dispatchers throughout the framework like before a checkpoint is loaded, before training starts, or after training finished. A full list of event dispatchers and their payload is listed in Table 3.2.

Event listeners watch the dispatchers and act on the payload. Features like differential privacy are implemented entirely with observers. As for user specified handlers, the `add_sigmoid` handler is the most useful ones, which adds a sigmoid activation to the output of a model. It should be attached to `before_criterion`, and `before_predict` dispatchers.

```
"observers": {
    "before_criterion": ["lib.core.observers.AddSigmoidHandler"],
    "after_predict": ["lib.core.observers.AddSigmoidHandler"]
},
```

CHAPTER

EIGHT

FEDERATED LEARNING (MILA)

Contents

- *Federated Learning (Mila)*
  - *Server Configurations*
    - *task_configuration_file (no default)*
    - *config_type: (no default)*
    - *executor_type: (no default)*
    - *aggregator_type (no default)*
    - *aggregator_options (default={})*
    - *target (default="localhost:8024")*
    - *rounds_count (default=10)*
    - *save_path (default="data/logs/server/")*
    - *start_point (default=null)*
    - *workers (default=2)*
    - *minimum_clients (default=2)*
    - *maximum_clients (default=100)*
    - *client_wait_time (default=10)*
    - *heartbeat_timeout (default=300)*
    - *use_secure_connection (default=false)*
    - *ssl_private_key (default="data/certificates/server.key")*
    - *ssl_cert (default="data/certificates/server.crt")*
    - *ssl_root_cert (default="data/certificates/rootCA.pem")*
    - *options*
    - *blacklist (default=[])*
    - *whitelist (default=[])*
    - *use_whitelist (default=false)*

- *Client Configurations*
  - `name (no default)`
  - `save_path (default="data/logs/client/")`
  - `heartbeat_frequency (default=60)`
  - `retry_timeout (default=1)`
  - `model_overwrites`
  - `config_type: (no default)`
  - `executor_type: (no default)`
  - `target (default="localhost:8024")`
  - `use_secure_connection (default=false)`
  - `ssl_private_key (default="data/certificates/client.key")`
  - `ssl_cert (default="data/certificates/client.crt")`
  - `ssl_root_cert (default="data/certificates/rootCA.pem")`
- *Mean CV Configuration*
  - `num_folds (no default)`
  - `seed (default = 42)`
  - `cfg_clients (default = none)`
  - `cfg_server (default = ")`
  - `cfg_dir (default = ")`
  - `log_level (default = "INFO")`
  - `save_results (default = False)`
  - `output_path (no default)`

Server and client processes are started similar to how models work. Each servicer and consumer requires a configuration file, which is fed to the entry point. In what follows, we describe the options for servers and clients separately.

8.1 Server Configurations

Server configuration files are expected to be in JSON format. The available options include:

8.1.1 `task_configuration_file (no default)`

Path to the model configuration file. This will be sent to each client along with the checkpoint.

8.1.2 `config_type: (no default)`

Module path to the configuration class (ie: “lib.config.Config”)

8.1.3 `executor_type: (no default)`

Module path to the executor class (ie: "run.Executor")

8.1.4 `aggregator_type (no default)`

Which aggregator to use? Options include: - PlainTorchAggregator - WeightedTorchAggregator - BenchmarkedTorchAggregator

more details <../module_details/aggregators>

8.1.5 `aggregator_options (default={})`

Input parameters for the aggregator (the expected value is a dictionary) "PlainTorchAggregator" does not expect anything; the "WeightedTorchAggregator" expects a "weights" options which is mapping between the client's "name" parameter and the expected wight. For example, if we have 2 clients named "tester1" and "tester2", this option could be:

```
"aggregator_options": {
    "weights":
        {
            "tester1": 0.67,
            "tester2": 0.33
        }
}
```

8.1.6 `target (default="localhost:8024")`

The gRPC service location URL (that is, the server address).

8.1.7 `rounds_count (default=10)`

How many rounds to perform

8.1.8 `save_path (default="data/logs/server/")`

Indicates where to save checkpoints received from clients and the aggregate models.

8.1.9 `start_point (default=null)`

Optionally, specify a checkpoint for the first round. If nothing is specified, clients will start training from scratch.

8.1.10 `workers (default=2)`

Maximum number of processes handling client requests

8.1.11 `minimum_clients (default=2)`

Minimum number of clients required to start federated learning. The server won't start the first round until this number is reached.

8.1.12 `maximum_clients (default=100)`

Maximum number of clients allowed to join the federated learning process.

8.1.13 client_wait_time (default=10)

Once the "minimum_clients" number is reach, the server will wait this many seconds for additional clients before the process starts. After this time expires, no new members will be allowed to join.

8.1.14 heartbeat_timeout (default=300)

Indicates how long to wait for a keep alive signal from clients before declaring them "dead".

8.1.15 use_secure_connection (default=false)

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

See here for more information about Secure connection <../tutorials/ssl_connection>

8.1.16 ssl_private_key (default="data/certificates/server.key")

gRPC secure communication private key

8.1.17 ssl_cert (default="data/certificates/server.crt")

gRPC secure communication SSL certificate

8.1.18 ssl_root_cert (default="data/certificates/rootCA.pem")

gRPC secure communication trusted root certificate

8.1.19 options

Additional gRPC options. "grpc.max_send_message_length" represents the maximum length of a sent message, and "grpc.max_receive_message_length" the maximum length of a received message. The default value for this option is:

```
"options": [
    ["grpc.max_send_message_length", 1000000000],
    ["grpc.max_receive_message_length", 1000000000],
    ["grpc.ssl_target_name_override", "localhost"]
]
```

8.1.20 blacklist (default=[])

A list of IP addresses which will be declined upon authentication.

8.1.21 whitelist (default=[])

A list of IP addresses which will be allowed to join the federated learning process.

If "use_whitelist" is True, these will be the only IP addresses allowed to join.

8.1.22 use_whitelist (default=false)

Enables whitelist filtering.

8.2 Client Configurations

Client configuration files are also expected to be in JSON format. The available options include:

8.2.1 name (no default)

A unique identifier for this client (could be a company name for example)

8.2.2 save_path (default="data/logs/client/")

Where to save checkpoints received from the client?

8.2.3 heartbeat_frequency (default=60)

Indicates, how often keep alive signals are sent to the server.

8.2.4 retry_timeout (default=1)

If a request fails because the server is under heavy load, we retry the connection after this many seconds.

8.2.5 model_overwrites

Used to override model configuration options. Generally, clients might want to change the path where local (model) checkpoints are stored. The default value for this option is:

```
{
"output_path": "data/logs/local/",
"epochs": 5
}
```

8.2.6 config_type: (no default)

Module path to the configuration class (ie: “lib.config.Config”)

8.2.7 executor_type: (no default)

Module path to the executor class (ie: “run.Executor”)

8.2.8 target (default="localhost:8024")

The gRPC service location URL (that is, the server address)

8.2.9 use_secure_connection (default=false)

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

See here for more information about Secure connection <../tutorials/ssl_connection>

8.2.10 ssl_private_key (default="data/certificates/client.key")

gRPC secure communication private key

8.2.11 `ssl_cert (default="data/certificates/client.crt")`

gRPC secure communication SSL certificate

8.2.12 `ssl_root_cert (default="data/certificates/rootCA.pem")`

gRPC secure communication trusted root certificate

8.3 Mean CV Configuration

8.3.1 `num_folds (no default)`

Number of folds to run the cross validation on.

8.3.2 `seed (default = 42)`

Seed to use for the experiment

8.3.3 `cfg_clients (default = none)`

List of path to client configuration to use during the experiment. This field needs to be used with *cfg_server*. This field can't be used with *cfg_dir*.

8.3.4 `cfg_server (default = '')`

Path to the server configuration. This field needs to be used with *cfg_clients*. This field can't be used with *cfg_dir*.

8.3.5 `cfg_dir (default = '')`

Path to a directory containing the client and server configs to use for the cross validation training. This field can't be used with both *cfg_clients* and *cfg_server*

8.3.6 `log_level (default = "INFO")`

Log level of the training possible values are `['CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET']`

8.3.7 `save_results (default = False)`

Whether to save the best checkpoints and the inference results for each fold.

8.3.8 `output_path (no default)`

Directory where the results will be stored.

CHAPTER

NINE

AGGREGATORS

Aggregators are responsible for network aggregation between all the clients. It is always a matter of merging the locally trained networks by individual clients after a given number of steps. Many methods exist and aggregation in federated learning is still an active area of research. The implemented methods in the framework are the following:

- `plain aggregation`
- `Weighted aggregation`
- `benchmark aggregation`
- `fedrox aggregation`

The aggregator is configurable in the server config file, here is an example:

```
{
    "task_configuration_file": "path/to/config.json",
    "aggregator_type": "mila.aggregators.WeightedTorchAggregator",
    "aggregator_options": {"weights": {"tester1": 0.8, "tester2": 0.2}},

    "config_type": "lib.core.config.Config",
    "executor_type": "run.Executor",

    "rounds_count": 50,
    "workers": 4,
    "minimum_clients": 2,
    "maximum_clients": 2,

    "client_wait_time": 300
}
```

9.1 Plain aggregation

Plain aggregation is invoked by the class `PlainTorchAggregator` and is the most basic aggregator in federated learning, it uses FedAverage. This aggregator considers all the clients even in the aggregation process no matter the data ratio between clients.

Plain aggregation doesn't need additional options.

9.2 Weighted aggregation

Weighted aggregator is defined in the `WeightedTorchAggregator`.

This aggregator scales the responsibility of each client depending on data distribution/ratio. Typically, a client with twice as many data than another client will be 2 times more important in the aggregation process. It is the most common application of FedAvg.

With this aggregator we need to provide the weights of each client additionally as aggregator_options:

```
"aggregator_options":
    {
        "weights":
            {
                "client_1": weight1,
                "client_2": weight2,
                ... : ...,
            }
    }
```

ex: `"aggregator_options": {"weights": {"tester1": 0.8, "tester2": 0.2}}`

9.3 Benchmark aggregation

This aggregator scale the responsibility of each client depending on the result of their respective evaluation score on the test set. Typically, a client with a local evaluation score twice better than another client will be 2 times more important in the aggregation process. This is especially useful when the test dataset used in evaluation is shared across clients.

With this aggregator we need to provide the following information as aggregator_options:

```
"aggregator_options":
    {
        "config_type": "config_type",
        "config_path": "config_path",
        "executor_type": "Executor_type",
        "target_metric": "target_metric",
    }
```

ex: `"aggregator_options": {"config_type": "lib.core.config.Config", "config_path": "path/to/config.json", "executor_type": "run.Executor", "target_metric": "rmse"}`

Note

The dataset used for evaluation is the test split provided in the model config at the given *config_path*. This means that this dataset needs to be present on the server node.

9.4 Fedrox aggregation

FedProx uses regularization to harmonize the individual training. In practice, FedProx penalizes each client for changing too much from the original network received by the server at each round. This is supposed to help convergence and reduce the dominance effect of some nodes.

Fedprox is configured differently than other aggregator and uses one of the previous aggregator like `PlainTorchAggregator` with an additional observer added in the model configuration:

```
"observers": {
        "after_criterion": [
            {
                "type": "add_fedprox_regularization",
                "mu": 1
            }
        ]
    },
```

Note

According to the authors of the paper, you might want to tune mu from {0.001, 0.01, 0.1, 0.5, 1}. There are no default mu values that would work for all settings.

CHAPTER TEN

CRITERIONS

The criterion option is used to specify the loss function. It is a dynamic option, however, it is not relying on abstractions.

Users can specify any native Torch or custom written loss functions:

```
"criterion": {
    "type": "torch.nn.BCEWithLogitsLoss"
},
```

We also provide a custom-built masked loss function, which wraps around a regular loss function, but ignores missing labels. Using the masked loss is easy with dependency injection:

```
"criterion": {
    "type": "lib.model.criterions.MaskedLoss",
    "loss": {"type": "torch.nn.BCEWithLogitsLoss"}
},
```

As a dynamic setting, of course, users can specify any additional input arguments for their loss function. For a list of all criterions, we point readers to the pytorch documentation on loss function.

CHAPTER ELEVEN

FEATURIZERS

```
"featurizers": [
    {
        "type": "graph",
        "inputs": ["smiles"],
        "outputs": ["ligand"],
        "descriptor_calculator": {"type": "rdkit"}
    }, {
        "type": "bag_of_words",
        "inputs": ["target_sequence"],
        "outputs": ["protein"],
        "should_cache": true,
        "vocabulary": [
            "A", "C", "D", "E", "F", "G", "H", "I", "K", "L", "M",
            "N", "P", "Q", "R", "S", "T", "V", "W", "Y", "X"
        ],
        "max_length": 3
    }
],
```

The above is an example configuration of a featurizer for protein-ligand affinity prediction

Featurizers prepare the input features for the training or inference. The expected input is a list, and therefore we can specify more than one featurizer per experiment. Featurizers are dynamic fields used to instantiate components extending `lib.data.featurizers.AbstractFeaturizer`.

As any dynamic field, the featurizer to be used is specified with the `type` option.

The supported featurizers are as follows:

`graph`:
For graph featurization. This featurizer expects a SMILES string for input, and the generated features can be used by graph architectures `graph_convolutional`, `message_passing`, or `triplet_message_passing`)

`circular_fingerprint`:
For fingerprint featurization. This featurizer expects a SMILES string for input, and the generated features can be used by the *linear* architecture.

`token`:
Similar to the one-hot encoder, but will tokenize a whole sentence (like an amino-sequence). The expected input is a sequence (string), and the output can be used by the *convolutional* architecture.

`bag_of_words`:
Performs an n-gram, bag-of-words featurization. The expected input is a sequence (string), and the generated features can be used by the *linear* architecture.

`one_hot_encoder`:
: Can one-hot encode categorical features. The expected input is a string. No architectures make direct use of this featurizer at this point, however, it could be used in a featurization pipeline to prepare inputs for other featurizers.

`transpose`:
: Is an intermediary featurizer. It expects a Torch Tensor as input and will output a transposed version of it. This of course can be done in a model directly, however, featurized outputs are cached, and can save valuable computational time.

`fixed`:
: Is an intermediary featurizer. It expects a single float as input and will return that value divided by a user specified value.

`tensor`:
: Featurizer for independant linear data. Featurizers for linear tabular type data. It return a torch type floating vector out of 1d data.

`tensor tabular`:
: Similar to *tensor* but concatenate all inputs in one 1d vector data.

Featurizers have custom arguments which can be configured, but all of them will have at least 3 parameters:

`inputs`:
: A list of input targets. These values should match the ones specified in the `input_column_names` option of the loader.

`outputs`:
: A list of output targets. These values should match the arguments expected by the network architectures. The list of arguments expected by each model is listed in Table 3.1.

`should_cache`:
: Whether we should perform sample level caching. Note that this is different from global caching, which is performed after featurization. When this option is set to true, each input will be cached in-memory and will not be processed again. This can be very helpful when we are dealing with many duplicates in a certain column which are expensive to compute, like amino-acid sequences. (defaults to `False`)

All featurizers will have these configurable options, and the `inputs` and `outputs` arguments are mandatory. However, most featurizers will have additional configurable options.

The `graph` featurizer supports:

`descriptor_calculator`:
: Specify how and what molecule level features to compute. The supported options include RDKit and mordred.

`allowed_atom_types`:
: Specify which atom types should be considered explicitly for (one-hot) encoding. Other atoms will be grouped as a general `other` feature. The default option includes: B, C, N, O, F, Na, Si, P, S, Cl, K, Br, and I.

In additional to symbol based encoding, the graph featurizer also encodes a one-hot atom degree encoding (from 0 to 10), one-hot implicit valence encoding (form 0 to 6), the formal charge, the number of radical electrons, one-hot encoded hybridization, an aromaticity flag, and the total number of Hydrogen atoms. By default, these amounts to 45 input features. If additional atom types are specified, the input features settings should reflect those changes.

For bond level features, we compute a bond type one-hot encoding (single, double, triple, or aromatic), a conjucation flag, a ring membership flag, a stereo configuration encoding. All these amount to 12 edge features.

The `circular_fingerprint` featurizer supports:

`fingerprint_size`:
: The number of bits (defaults to `2048`)

`radius`:
: Defaults to `2`

The `one_hot_encoder` featurizer requires:

`classes`:
: A list of all possible tokens (ie: `male`, `female`)

The `token` featurizer supports:

`vocabulary`:
: A list of all possible tokens (ie: `A`, `C`, `T`, `G`)

`max_length`:
: The length of the largest sequence (smaller sequences will be 0-padded)

`separator`:
: A separator for the tokens. If an empty string `""` is specified, the string will be split character-by-character (defaults to `""`)

The `bag_of_words` featurizer supports:

`vocabulary`:
: A list of all possible tokens (ie: `A`, `C`, `T`, `G`)

`max_length`:
: The length of the largest sequence (smaller sequences will be 0-padded)

The `fixed` featurizer requires:
`value`: the value to divide the input by

The `tensor` and `tensor_tabular` featurizer doesn't require additional feature.

CHAPTER TWELVE

LOADERS

The following is an example configuration of a CSV loader for the AMES dataset:

```
"loader": {
    "type": "csv",
    "input_path": "data/input/tox21/raw/tox21.csv",
    "input_column_names": ["smiles"],
    "target_column_names": [
        "NR-AR", "NR-AR-LBD", "NR-AhR", "NR-Aromatase",
        "NR-ER", "NR-ER-LBD", "NR-PPAR-gamma", "SR-ARE",
        "SR-ATAD5", "SR-HSE", "SR-MMP", "SR-p53"
    ]
},
```

The loader option is used to configure how raw data files are loaded. It is a dynamic field used to instantiate components which extend `lib.data.loaders.AbstractLoader`.

We support 3 loaders at this time, which are specified in the `type`:

- **csv**: For comma-separated values
- **excel**: For Excel spreadsheets (only one spreadsheet can be loaded at a time)
- **sdf**: For SDF files (where the whole dataset is stored in a single SDF file)

The additional arguments are identical for the most part at this time. One exception is the **excel** loader, which accepts a numeric `sheet_index` to specify which sheet to load (numbering starts from 0 for the first sheet in the file). Otherwise, all loaders are expected to receive 3 fields:

`input_path`:
  The relative or absolute path to the input file

`input_column_names`:
  A list of columns or properties which should be mapped as input features

`target_column_names`:
  A list of columns or properties which should be mapped as output features

CHAPTER

THIRTEEN

MODELS

The model option is used to specify which model architecture to use and the options of that model. It is a dynamic field and used to instantiate torch modules which extend `lib.model.architectures.AbstractNetwork`.

As a dynamic field, the "type" argument is used to specify which class to use. At the moment, we support 6 architectures as input for the "type" option:

- `graph_convolutional`
- `message_passing`
- `triplet_message_passing`
- `linear`
- `convolutional`
- `protein_ligand`
- `SimpleMLP`

The following is an example of configuring a Graph Convolutional Network

```
"model": {
    "type": "graph_convolutional",
    "in_features": 45,
    "out_features": 1,
    "hidden_features": 160,
    "dropout": 0.1,
    "layer_type": "torch_geometric.nn.GCNConv",
    "layers_count": 3,
    "molecule_features": 17,
    "is_residual": 1,
    "norm_layer": "lib.model.layers.BatchNorm"
}
```

13.1 Graph Convolutional Networks

The `graph_convolutional` is by far the most configurable architecture. Customizable arguments for this model are as follows:

`in_features:`
The number of input features. This value will depend on the options configured for graph featurizer, but should be `45` by default.

`hidden_features:`
Number of hidden features used in the graph convolutional operation.

`out_features`:
    The number of output features. This should coincide with the number of concurrent tasks we are trying to predict (ie: 12 for Tox21, and 1 for AMES).

`molecule_features`:
    How many molecule level features are we expecting? This should be set to `17` for RdKit descriptors and `1613` for Mordred descriptors.

`dropout`:
    The dropout rate

`layer_type`:
    The graph convolutional operator (defaults to `torch_geometric.nn.GCNConv`). We leverage implementations from Pytorch Geometric and support many layers listed in torch_geometric.nn. Layers which have been tested to work are as follows:

- `torch_geometric.nn.GCNConv`
- `torch_geometric.nn.ChebConv`
- `torch_geometric.nn.SAGEConv`
- `torch_geometric.nn.GraphConv`
- `torch_geometric.nn.ARMAConv`
- `torch_geometric.nn.LEConv`
- `torch_geometric.nn.GENConv`
- `torch_geometric.nn.ClusterGCNConv`
- `torch_geometric.nn.FeaStConv`
- `torch_geometric.nn.GATConv`
- `torch_geometric.nn.TAGConv`
- `torch_geometric.nn.SGConv`
- `lib.model.layers.GINConvolution`
- `lib.model.layers.TrimConvolution`

`layers_count`:
    The number of graph layers to use (defaults to `2`)

`is_residual`:
    Whether to apply a residual connection to the graph operation (defaults to `True`)

`norm_layer`:
    Which normalization layer to apply after the graph operation (defaults to `None`). The available options are:

- `None`
- `lib.model.layers.BatchNorm`
- `lib.model.layers.GraphNorm`
- `torch_geometric.nn.LayerNorm`

`activation`:
    The activation function to use (defaults to `torch.nn.ReLU`). For a list of options, please check the PyTorch documentation on non-linear activations.

edge_features:
The number of edge features (defaults to 0). At the moment, the only working options are 0 or 12. If set to 12, edge features will be concatenated with atom features.

13.2 Message Passing Neural Networks

The message_passing architecture accepts the following options:

in_features:
The number of input features. This value will depend on the options configured for graph featurizer, but should be "45" by default.

hidden_features:
Number of hidden features used in the graph convolutional kernel.

out_features:
The number of output features.

edge_features:
The number of edge features. At the moment, the only working option is 12.

edge_hidden:
The number of hidden features for the edge block.

steps:
Number of processing steps to run

dropout:
The dropout rate (defaults to 0)

aggregation:
The aggregation scheme to use. (Options are: add, mean, or max) (defaults to add)

set2set_layers:
Number of recurrent layers to use in the global pooling operator (defaults to 3)

set2set_steps:
Processing steps for the global pooling operator (defaults to 6)

13.3 Triplet Message Passing Networks

The triplet_message_passing architecture accepts the following options:

in_features:
The number of input features. This value will depend on the options configured for graph featurizer, but should be 45 by default.

hidden_features:
Number of hidden features used in the graph convolutional kernel.

out_features:
The number of output features.

edge_features:
The number of edge features. At the moment, the only working option is 12.

layers_count:
The number of Triplet Message Passing layers to use

dropout:
The dropout rate (defaults to 0)

`set2set_layers`:
: Number of recurrent layers to use in the global pooling operator (defaults to `1`)

`set2set_steps`:
: Processing steps for the global pooling operator (defaults to `6`)

13.4 Linear Networks

The `linear` architecture is a simple Shallow Neural Network with 2 linear layers.

`in_features`:
: The number of input features

`hidden_features`:
: The number of hidden features

`out_features`:
: The number of output features

`activation`:
: The activation type (defaults to `torch.nn.ReLU`)

13.5 SimpleMLPNetwork

The `SimpleMLPNetwork` s a classic MLP Network used for tabular data. It is a succession of `n_layer` fully connected layer.

`in_features`:
: The number of input features

`hidden_features`:
: The number of hidden features

`out_features`:
: The number of output features

`n_layer`:
: The number of fully connected layer

`dropout`:
: Dropout for each block. Dropout is used for better generalization.

13.6 Convolutional Networks

The `convolutional` architecture is a simple Convolutional Network with a single 1D convolutional layer, and a max pooling layer, followed by a linear block as describe above:

`in_features`:
: The number of input features

`hidden_features`:
: The number of hidden features

`out_features`:
: The number of output features

13.7 Protein Ligand

The protein_ligand is a composite architecture. That is, it takes 2 other modules as input, one for the ligand and one for the protein features. Dependency injection again makes our work easy.

The following is an example configuration for Protein-Ligand Architecture

```
"model": {
    "type": "protein_ligand",
    "protein_module": {
        "type": "linear",
        "in_features": 9723,
        "hidden_features": 160,
        "out_features": 16
    },
    "ligand_module": {
        "type": "graph_convolutional",
        "in_features": 45,
        "out_features": 16,
        "hidden_features": 192,
        "dropout": 0.0,
        "layer_type": "torch_geometric.nn.GENConv",
        "layers_count": 5,
        "molecule_features": 17,
        "is_residual": 0,
        "norm_layer": "lib.model.layers.BatchNorm"
    },
    "hidden_features": 32,
    "out_features": 3
}
```

protein_module:
A list set of options for the protein module. Supported options include a *convolutional* architecture for tokenized inputs, or a *linear* architecture for bag-of-words featurized inputs.

ligand_module:
A list set of options for the ligand module.

Supported options include a graph_convolutional, message_passing, or triplet_message_passing architecture for graph featurized ligands, or a linear architecture for circular fingerprints.

hidden_features:
The number of hidden features

out_features:
The number of output features

After the individual blocks are passed through, the outputs are concatenated and passed through a final linear block.

CHAPTER

FOURTEEN

SPLITTERS

The last of the major abstraction based dynamic settings, the "splitter" specifies how to split the dataset. These options are used to instantiate components extending `lib.data.splitters.AbstractSplitter`.

An example of configuration for a stratified splitter:

```
"splitter": {
    "type": "stratified",
    "seed": 42,
    "target_name": "label",
    "splits": {"train": 0.8, "test": 0.2}
},
```

As a dynamic option, splitter classes are specified using the "type" argument. We have the following splitter types:

`index`:
: For index-based splitting

`random`:
: For random splits

`stratified`:
: For stratified splits based on a certain output column/property

`categorical`:
: For splitting with each split having distinct element of the specified category.

14.1 Shared argument

`splits`

All splits require a `splits` argument in a dictionary (key-value) format. The keys denote the name of the split and can be any users-specified value (we generally used `train` and `test`). The values represent the proportions for each split and should add up to 1. Users can specify any number of splits, not only 2.

In most cases when using the `mean_cv` job the split are overwritten for the right number of folds.

`seed (default = 42)`

The `random` / `stratified` and `categorical` splitters also require a `seed` argument which is used to set the random state for reproducible splits.

14.2 Stratified

`target_name`

The `stratified` split requires a `target_name` argument, which should be an entry of the loader's `target_column_names` argument. The split will be performed based on this column. If the `target_name` contains continuous values, they can be grouped into a number of bins using quantile-based discretization (ie: equal-sized buckets based on rank). Users can specify the number of bins using the `bins_count` argument.

`is_target_input (default = False)`

In case the `target_name` is used as an input in the loader.

`rewrite (default = False)`

If `is_target_input` is True by settings `rewrite` to True the feature will be deleted from the input and so will only be used for splitting and not during training.

14.3 Categorical

`cat_name`

The name of the input_feature to use for the splitting.

Warning

There is two case in this scenario.

Case 1:

For training with `nims-federated-learning-base` or in the case you are using `mean_cv` task of `nims-federated-learning` the splitting occurs has other models.

Case 2:

In case of standard training with `nims-federated-learning` using the `subset distribution` argument of model configuration is not possible to split the data between client since the splits needs to be made base on the category information.

So in this case `subset distribution` is not useful. But you need to provide and `id` and the name of the split must be `client_{id}`.

For example let's look into config made for the multi-step experiment:

```
// client2.json
// notice we only use subset for the id, here 1.
{
    "name": "tester2",
    "config_type": "lib.core.config.Config",
    "save_path": "data/logs/client/tester2/",
    "executor_type": "run.Executor",
    "model_overwrites": {
        "output_path": "data/logs/local/tester2/",
        "epochs": 10,
        "subset": {
            "id": 1
        }
    }
}
```

```
// Here we must provide the distribution for each client, client2 will have client_
↪{id}
// So client_1 value it will use 8% of the dataset (10% of the training data)
{
...
"splitter": {
    "type": "Categorical",
    "cat_name": "Reference Code",
    "splits": {
        "client_0": 0.48,
        "client_1":0.08,
        "client_2":0.08,
        "client_3":0.08,
        "client_4":0.08,
        "test": 0.2
    },
    "seed": 42
},
...
}
```

This is a constraint but since the categorical set up is not possible to use outside testing scenario with a full dataset it is possible to fix all those parameters.

CHAPTER

FIFTEEN

TRANSFORMERS

Transformers are very similar to featurizers, expect they are applied to output features, and are generally used to normalize values. Transformers are dynamic fields used to instantiate components extending `lib.data.transformers.AbstractTransformer`.

Each transformer implements an "apply" and a "reverse" operation, which converts the values back after inference (for prediction only, does not apply to metrics). Similar to featurizers, multiple transformers can be used in the same experiment

The following is an example of how to configure transformers

```
"transformers": [
    {"type": "log_normalize", "targets": [0, 2]},
    {"type": "standardize", "target": 1, "mean": 2.1863357142857143, "std": 1.
↪203003713387104}
],
```

As a dynamic field, the transformer type is again specified with the "type" argument:

`log_normalize`:
Converts the target to its log value. This transformer expects a `targets` argument, which is a list of (0-based) indices mapping to the "target_column_names" argument of the loader.

`min performs min-max normalization`:
This transformer has a `target` argument similar to the log transformer, however, in this case, we expect a single value instead of a list. The "minimum" and "maximum" arguments are also mandatory, which should reflect the smallest and largest values recorded for the column/property.

`fixed`:
Divides the target values by a fixed number. This transformer expects a `targets` argument, which is a list in this case. The "value" argument is used to specify the value used for the division.

`standardize`:
Performs z-score normalization. This transformer has a `target` argument which is a single 0-based index (not a list). The "mean" and "std" options are also mandatory which denote the average value and the standard deviation for the target.

`cutoff`:
Converts a continuous value to a discrete one, using a user-specified cutoff. Note that this operation is destructive and cannot be reversed. We expect a `target` argument with a single 0-based index, and a "cutoff" argument denoting the threshold for binarization.