ml-experiment

Release 0.1

Antonio Molner, Alberto Guillen

Jul 03, 2020

CONTENTS:

CHAPTER

ONE

INSTALLATION

ml-experiment supports Python 3.6 and above.

We use Ray for hyperparameter optimization, so as Ray currently supports MacOS and Linux only, Windows support will be available as soon as this framework supports it.

We recommend installing ml-experiment using Pip:

```
pip install ml-experiment
```

CHAPTER

TWO

RUNNING YOUR FIRST EXPERIMENT

2.1 Quickstart

2.1.1 1. Instrumenting a python script

```
from ml_experiment import job

import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

SEED = 1234

@job # ADD THIS DECORATOR
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    np.random.seed(SEED)
    iris = load_iris()
    X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target, random_
↪state=SEED)
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}


if __name__ == '__main__':
    main()
```

2.1.2 2. Defining a configuration file

Defining a configuration file is straightforward, we just need to create a YAML/JSON file and specify the name of the experiment, its parameters, and the file to execute.

```
name: "SVM #1"

params:
  C: 10
  kernel: poly
  gamma: auto
  degree: 3
```

(continues on next page)

(continued from previous page)

```
run:
  - examples/scripts/train_svm.py
```

2.1.3 3. Executing the experiment

Finally, once we have an instrumented Python script and a config file, we can execute the job as follows:

```
ml-experiment --config_file examples/experiments/svm.yaml
```

2.2 Running jobs in a Docker container

```
docker_config:
  image: image_name:tag
```

```
docker_config:
  dockerfile: path/to/dockerfile
```

2.3 Adding callbacks

Callbacks are an important aspect in almost any ML/DL framework. Callbacks allow one to hook into the process and react to some event happening.

ml-experiment offers a simple callback system based on the class Callback. In order to implement a custom callback, all it takes is implementing that abstract class.

Once we've implemented a custom callback class, we can add an instance of it as *callbacks* argument of job

The module notifiers contains some predefined callbacks for notifications. It

```
from ml_experiment import job
from ml_experiment.callbacks.notifiers import DesktopNotifier

import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

SEED = 1234

# ADD THIS
@job(callbacks=[DesktopNotifier()])
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    np.random.seed(SEED)
    iris = load_iris()
    X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target, random_
↪state=SEED)
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
```

(continues on next page)

(continued from previous page)

```
    return {'val_accuracy': accuracy}


if __name__ == '__main__':
    main()
```

CHAPTER

THREE

HYPERPARAMETER TUNING

3.1 From experiments to group of experiments

```yaml
name: "SVM"
kind: 'group'
num_trials: 10

param_space:
  C: loguniform(0.01, 1000)
  kernel: choice(['rbf', 'poly', 'linear'])
  gamma: choice(['scale', 'auto'])
  degree: range(2, 5)

params:
  C: 1.0

metric:
  name: val_accuracy
  direction: maximize

run:
  - examples/scripts/train_svm.py
```

3.1.1 Configuring the Ray cluster

```yaml
resources_per_worker:
  cpu: 0.25
  gpu: 0.5

ray_config:
  num_cpus: 4
  num_gpus: 1
```

NOTE: Docker integration and Ray integration are incompatible for the moment. So, Docker is not supported for running groups of experiments.

3.1.2 Pruning unpromising trails

```yaml
sampler: tpe
pruner: hyperband
```

3.2 Sharing data across multiples processes

When we are executing a group of experiments multiples processes are created (one for each experiment), as a consequence of this, when we load, compute, or transform some data, we need to execute that computation multiples times. To avoid that, we propose a solution inspired by Pytorch data loaders.

In our case, we create a class inherited from ml_experiment.DataLoader and define the load_data method. This method will be executed **only** by the master node all of the other processes will have a copy of the data generated by that method.

After we have created a custom DataLoader, we need to pass it as *data_loader* argument to job and after that, we can use it as it we used any other class.

> The shared data will be stored in the Plasma Object Store of ray, so you should take into account its limitations: Ray Serialization

```python
from ml_experiment import job, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler


SEED = 1234


class MyDataLoader(DataLoader):
    @classmethod
    def load_data(cls):
        iris = load_iris()
        X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target,␣
↪random_state=SEED)
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)
        return X_train, X_val, y_train, y_val


@job(data_loader=MyDataLoader)
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    X_train, X_val, y_train, y_val = MyDataLoader.load_data()
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}
```

3.3 Accessing the Trial instance to model a complex parameter space

Sometimes, it may be necessary to access the optuna.Trial object of the current experiment so we can generate a more complex hyperparameter space. To do so, we just need to the following:

```
from ml_experiment import job, Trial

@job
def main(param1, ..., paramN):
    trial = Trial.get_current()
    ...
```

If you're running a single experiment instead of a group, Trial.get_current will raise an exception.

CHAPTER

FOUR

YAML/JSON SPECIFICATION

Configuration files are validated and parsed into Python objects. Therefore, config files have the same structure as the Python models defined on experiment_specification/Models Reference. Feel free to check them in case of doubt.

4.1 Experiment Definition

```yaml
name: str # Required
kind: experiment # Optional. This is the default value

# Optional. Defaults to an empty dict
params:
  param1: int | str | list | dict
  param2: ...
  ...
  paramN: ...

# Optional. If not specified, the job will be run in the host environment (without␣
↪Docker).
docker_config:
  image: str
  dockerfile: path/to/dockerfile
  context: path/to/context/directory
  args: dict

# Optional. If not specified, the job will be run in a local environment (without␣
↪Ray).
# In any case, only one process will be spawned.
# Any other entry of this dictionary will be passed as it is to Ray.init,
# so you can fully configure the job execution.
# More information about the parameters you can use here:
# https://docs.ray.io/en/master/package-ref.html#ray.init
ray_config:
  address: localhost | master_node_address

run: # Required
  - path/to/script1.py
  ...
  - path/to/scriptN.py
```

4.2 Group Definition

```yaml
name: str   # Required.
kind: group # Required. This line should be specified for ml-experiment CLI to know␣
↪what type of job is this
sampler: str # Optional.
pruner: str  # Optional.
timeout_per_trial: positive float # Optional
resources_per_worker: # Optional
  cpu: positive float # Required (only if the parent is specified)
  gpu: positive float # Optional.


# Optional. Defaults to an empty dict
param_space:
  param1: distribution(x1, x2, ..., xN)
  ...
  paramN: distribution(x1, x2, ..., xN)

  # Distributions can also be used inside a list.
  # The behavior is to sample a random value for each position
  otherParam:
    - distribution(x1, ..., xN]
    ...
    - distribution(x1, ..., xN)

# Optional. Defaults to an empty dict
params:
  otherParam1: int | str | list | dict
  ...
  otherParamN: ...

# Optional. If not specified, the job will be run in a local Ray cluster.
# Any other entry of this dictionary will be passed as it is to Ray.init,
# so you can fully configure the job execution.
# More information about the parameters you can use here:
ray_config:
  address: localhost | master_node_address

run:
  - path/to/script1
  ...
  - path/to/scriptN
```

4.3 Parameter space distributions

Generates a random integer from min to max with an specific step.

```
range(min: int, max: int, step: int)
```

Generates a random integer from min to max (same as range with step = 1)

```
randint(min: int, max: int)
```

A uniform distribution in the log domain.

```
loguniform(min: int, max: int)
```

A uniform distribution in the linear domain.

```
uniform(min: int, max: int)
```

A categorical distribution based on the values provided. The sample parameter will be selected randomly (with uniform probability) among the provided values.

```
choice([value1: any, value2: any, ..., valueN: int])
```

4.4 Models Reference

class `ml_experiment.config.models.`**`ExperimentConfig`**`(`***data*`)`

Bases: `ml_experiment.config.models.JobConfig`

`docker_config = None`

`kind = None`

`name = None`

`params = None`

`ray_config = None`

`run = None`

class `ml_experiment.config.models.`**`GroupConfig`**`(`***data*`)`

Bases: `ml_experiment.config.models.JobConfig`

`metric: Optional[Metric] = None`

`num_trials: PositiveInt = None`

`param_space: Dict[str, Union[ParamDistribution, List[ParamDistribution]]] = None`

`pruner: Optional[PrunerEnum] = None`

`resources_per_worker: WorkerResourcesConfig = None`

`sampler: Optional[SamplerEnum] = None`

`timeout_per_trial: Optional[PositiveFloat] = None`

class `ml_experiment.config.models.`**`RayConfig`**`(`***data*`)`

Bases: `pydantic.main.BaseModel`

`class Config`

Bases: `object`

`extra = 'allow'`

`address: Optional[str] = None`

`classmethod convert_localhost``(`*v*`)`

Parameters v (*str*) –

class `ml_experiment.config.models.`**`JobTypes`**

Bases: `enum.Enum`

An enumeration.

EXPERIMENT = 'experiment'

GROUP = 'group'

JOB = 'job'

class ml_experiment.config.models.**Metric**(***data*)

Bases: pydantic.main.BaseModel

direction: OptimizationDirection = None

name: str = None

class ml_experiment.config.models.**DockerConfig**(***data*)

Bases: pydantic.main.BaseModel

args: Dict = None

classmethod check_dockerfile_and_image(*values*)

context: Optional[DirectoryPath] = None

dockerfile: Optional[FilePath] = None

image: Optional[str] = None

class ml_experiment.config.models.**WorkerResourcesConfig**(***data*)

Bases: pydantic.main.BaseModel

cpu: PositiveFloat = None

gpu: PositiveFloat = None

class ml_experiment.config.models.**PrunerEnum**

An enumeration.

hyperband = 'hyperband'

median = 'median'

percentile = 'percentile'

sha = 'sha'

class ml_experiment.config.models.**SamplerEnum**

An enumeration.

random = 'random'

skopt = 'skopt'

tpe = 'tpe'

CHAPTER

FIVE

PACKAGE REFERENCE

5.1 ml_experiment

@ml_experiment.**job**(*func=None*, ***, *callbacks=None*, *autologging_backends=None*, *optimization_metric=None*, *data_loader=None*, *log_seeds=True*, *log_system_info=True*, *delete_if_failed=False*, ***kwargs*)

Experiment decorator.

This decorator must be used as a wrapper for the main function of the experiment. It handles the tracking of the following information: - Parameters - Metrics - Artifacts - System information - Randomness (Numpy, Pytorch and TF seeds)

It also handles the job execution on clusters and the hyperparameter optimization logic.

Parameters

- **func** (*Optional[Callable]*) – Experiment main function
- **callbacks** (*Optional[Iterable[ml_experiment.callbacks.core.Callback]]*) – List of callbacks to notify when some event occur
- **autologging_backends** (*Union[List[ml_experiment.mlflow.AutologgingBackend], ml_experiment.mlflow.AutologgingBackend, None]*) – List of frameworks whose autologging functionality will be enabled. To specify a supported framework you need to use the AutologgingBackend enum.
- **optimization_metric** (*Union[ml_experiment.config.models.Metric, str, None]*) – Metric to optimize. It is mandatory when running a group job, and it will be ignored when running a single experiment. The name of the metric should be the same as one of the keys that the experiment function returns.
- **data_loader** (*Optional[ml_experiment.experiments.DataLoader]*) – Custom data loader class (not instance). It is necessary to specify this argument when using a DataLoader to share data across multiples processes.
- **log_seeds** (*bool*) – If true, it will log the seed of Numpy, Pytorch, or Python random's generator when the corresponding function to set the seed is called. Eg. when calling numpy.random.sed(…)
- **log_system_info** (*bool*) – Whether or not the system information, CPU, GPU, installed packages…, etc, should be logged
- **delete_if_failed** (*bool*) – If true, the experiment information will be removed in case of failure.

Returns The wrapped function

class ml_experiment.**DataLoader**

Base class for Data loaders.

The main purpose of data loaders is to provide an easy way to share data across processes when running a group of experiments, also known as hyperparameter tuning.

When this abstract class is implemented (using a subclass) and that subclass is added as the argument *data_loader* to the experiment main function decorator, a shared resource will be created. This shared resource is the result of executing the implemented function, load_data. The key point here is that the load_data function will be only called once by the master process and then its result will be shared among the rest workers. In this way, we can avoid expensive computation being duplicated for each worker.

The shared data will be stored in the Plasma Object Store of ray, so you should take into account its limitations: https://docs.ray.io/en/latest/serialization.html

classmethod load_data()

class ml_experiment.**Trial**

This class makes the current Optuna Trial object accessible.

It can be used to model complex hyperparameter spaces. More information here: https://optuna.readthedocs.io/en/latest/tutorial/configurations.html

classmethod get_current()

Return type optuna.trial.Trial

class ml_experiment.**AutologgingBackend**

An enumeration.

FASTAI = 'fastai'

KERAS = 'keras'

LIGHTGBM = 'lightgbm'

TENSORFLOW = 'tensorflow'

XGBOOST = 'xgboost'

5.2 ml_experiment.callbacks

class ml_experiment.callbacks.**Callback**

Base class for callbacks that want to react to fired events.

To create a new type of callback, you'll need to inherit from this class, and implement one or more methods as required for your purposes. Arguably the easiest way to get started is to look at the source code for some of the pre-defined ones.

on_info_logged(*config*, *metrics*, *artifacts*, ***kwargs*)

This method will be execute every time the metrics and artifacts are logged. It will be called at least one time for every experiment.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) – The configuration object contains all the information regarding how the experiment is executed
- **metrics** (*Dict[str, Any]*) – The metrics dictionary returned by the main function
- **artifacts** (*Dict[Dict, Any]*) – The artifacts dictionary returned by the main function

on_job_end (*config*, *exception*)

This method will be execute once the experiment has ended :param config: The configuration object contains all the information regarding how the experiment is executed :param exception: If not none, it means that the experiment has finished due to an error. This param contains that error.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **exception** (*Optional[Exception]*) –

on_job_start (*config*, ***kwargs*)

This method will be execute once the experiment has started :param config: The configuration object contains all the information regarding how the experiment is executed

Parameters **config** (*ml_experiment.config.models.JobConfig*) –

on_trial_end (*config*, *trial*, *metric*, *exception*)

Only for groups of experiments.

This method will be execute once a trial has been finished. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param metric: If everything when fine, it will contain the value of the optimization metric :param exception: If not none, it means that the trial has finished due to an error. This param contains that error.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **trial** (*optuna.trial.Trial*) –
- **metric** (*Optional[float]*) –
- **exception** (*Optional[Exception]*) –

on_trial_start (*config*, *trial*, *sampled_params*)

Only for groups of experiments.

This method will be execute once a trial has been started. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param sampled_params: The randomly selected parameters

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **trial** (*optuna.trial.Trial*) –
- **sampled_params** (*dict*) –

class ml_experiment.callbacks.notifiers.**NotifierBase**

Base class for notifiers

To create a new type of notifier, you'll need to inherit from this class, and implement one or more methods as required for your purposes. Specifically, the send_message method should be override. Arguably the easiest way to get started is to look at the source code for some of the pre-defined ones.

Parameters **send_message_for_trials** – If true, messages will be send for trial-related events.

on_job_end (*config*, *exception*)

This method will be execute once the experiment has ended :param config: The configuration object contains all the information regarding how the experiment is executed :param exception: If not none, it means that the experiment has finished due to an error. This param contains that error.

on_job_start(*config*, ***kwargs*)
This method will be execute once the experiment has started :param config: The configuration object contains all the information regarding how the experiment is executed

on_trial_end(*config*, *trial*, *metric*, *exception*)
Only for groups of experiments.

This method will be execute once a trial has been finished. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param metric: If everything when fine, it will contain the value of the optimization metric :param exception: If not none, it means that the trial has finished due to an error. This param contains that error.

on_trial_start(*config*, *trial*, *sampled_params*, ***kwargs*)
Only for groups of experiments.

This method will be execute once a trial has been started. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param sampled_params: The randomly selected parameters

abstract send_message(*msg*)
This method should be override by your custom logic

Parameters msg (`str`) – The preprocessed messaged generated from the experiment information

Return type None

class ml_experiment.callbacks.notifiers.**TelegramNotifier**(*token*, *chat_id*)
Callback that sends a message through a Telegram Bot.

It requires a token and chat_id which can be get following these instructions: https://core.telegram.org/bots

Parameters

- **token** – Telegram Bot Token.
- **chat_id** – Telegram group id. More info on how to get this id here.

class ml_experiment.callbacks.notifiers.**DesktopNotifier**
Callback that sends a desktop notification when an event happens.

It works out-of-the-box for Linux and OS X. For Windows it is necessary to install the win10toast package.

class ml_experiment.callbacks.notifiers.**SlackNotifier**(*webhook_url*, *channel*, *username*)
Callback that sends a message to an specific Slack channel when an event happens.

It requires a webhook URL, in order to generate that URL you should follow these instructions

Parameters

- **webhook_url** – Theb webhook generated from Slack Apps
- **channel** – Slack channel name
- **username** – The username that will appear on the slack messages

class ml_experiment.callbacks.notifiers.**EmailNotifier**(*sender_email*, *recipient_emails*)
Callback that sends a email to a recipient or list of recipients when an event happens.

This service relies on Yagmail, so you'll need to setup a Gmail account and follow the library instructions.

5.3 ml_experiment.integrations

All these classes are imported from the Optuna package. For more information of how to use, please take a look at the official documentation here.

class ml_experiment.integrations.**KerasPruningCallback**(*trial*, *monitor*)

Keras callback to prune unpromising trials.

Example

Add a pruning callback which observes validation losses.

```
model.fit(X, y, callbacks=[KerasPruningCallback(trial, 'val_loss')])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., val_loss and val_acc. Please refer to keras.Callback reference for further details.

class ml_experiment.integrations.**TensorFlowPruningHook**(*trial*, *estimator*, *metric*, *run_every_steps*, *is_higher_better=None*)

TensorFlow SessionRunHook to prune unpromising trials.

Example

Add a pruning SessionRunHook for a TensorFlow's Estimator.

```
pruning_hook = TensorFlowPruningHook(
    trial=trial,
    estimator=clf,
    metric="accuracy",
    is_higher_better=True,
    run_every_steps=10,
)
hooks = [pruning_hook]
tf.estimator.train_and_evaluate(
    clf,
    tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=500, hooks=hooks),
    eval_spec
)
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **estimator** – An estimator which you will use.
- **metric** – An evaluation metric for pruning, e.g., accuracy and loss.
- **run_every_steps** – An interval to watch the summary file.
- **is_higher_better** – Please do not use this argument because this class refers to StudyDirection to check whether the current study is minimize or maximize.

class ml_experiment.integrations.**TFKerasPruningCallback**(*trial*, *monitor*)
tf.keras callback to prune unpromising trials.

This callback is intend to be compatible for TensorFlow v1 and v2, but only tested with TensorFlow v1.

Example

Add a pruning callback which observes validation losses.

```
model.fit(x, y, callbacks=[TFKerasPruningCallback(trial, 'val_loss')])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., val_loss or val_acc.

class ml_experiment.integrations.**XGBoostPruningCallback**(*trial*, *observation_key*)
Callback for XGBoost to prune unpromising trials.

Example

Add a pruning callback which observes validation errors to training of an XGBoost model.

```
pruning_callback = XGBoostPruningCallback(trial, 'validation-error')
bst = xgb.train(param, dtrain, evals=[(dtest, 'validation')],
                callbacks=[pruning_callback])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **observation_key** – An evaluation metric for pruning, e.g., validation-error and validation-merror. Please refer to eval_metric in XGBoost reference for further details.

class ml_experiment.integrations.**LightGBMPruningCallback**(*trial*, *metric*, *valid_name='valid_0'*)
Callback for LightGBM to prune unpromising trials.

Example

Add a pruning callback which observes validation scores to training of a LightGBM model.

```
param = {'objective': 'binary', 'metric': 'binary_error'}
pruning_callback = LightGBMPruningCallback(trial, 'binary_error')
gbm = lgb.train(param, dtrain, valid_sets=[dtest], callbacks=[pruning_callback])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **metric** – An evaluation metric for pruning, e.g., binary_error and multi_error. Please refer to LightGBM reference for further details.

- **valid_name** – The name of the target validation. Validation names are specified by `valid_names` option of train method. If omitted, `valid_0` is used which is the default name of the first validation. Note that this argument will be ignored if you are calling cv method instead of train method.

class ml_experiment.integrations.**PyTorchIgnitePruningHandler**(*trial*, *metric*, *trainer*)

PyTorch Ignite handler to prune unpromising trials.

Example

Add a pruning handler which observes validation accuracy.

```
evaluator = create_supervised_evaluator(model,
                                        metrics={'accuracy': Accuracy()},
                                        device=device)
handler = PyTorchIgnitePruningHandler(trial, 'accuracy', trainer)
evaluator.add_event_handler(Events.COMPLETED, handler)

@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    evaluator.run(val_loader)
```

Parameters

- **trial** – A `Trial` corresponding to the current evaluation of the objective function.
- **metric** – A name of metric for pruning, e.g., `accuracy` and `loss`.
- **trainer** – A trainer engine of PyTorch Ignite. Please refer to ignite.engine.Engine reference for further details.

class ml_experiment.integrations.**PyTorchLightningPruningCallback**(*trial*, *monitor*)

PyTorch Lightning callback to prune unpromising trials.

Example

Add a pruning callback which observes validation accuracy.

```
trainer.pytorch_lightning.Trainer(
    early_stop_callback=PyTorchLightningPruningCallback(trial, monitor='avg_val_
↪acc'))
```

Parameters

- **trial** – A `Trial` corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., `val_loss` or `val_acc`. The metrics are obtained from the returned dictionaries from e.g. `pytorch_lightning.LightningModule.training_step` or `pytorch_lightning.LightningModule.validation_end` and the names thus depend on how this dictionary is formatted.

class ml_experiment.integrations.**FastAIPruningCallback**(*learn*, *trial*, *monitor*)

FastAI callback to prune unpromising trials for fastai.

Note: This callback is for fastai<2.0, not the coming version developed in fastai/fastai_dev.

Example

Add a pruning callback which monitors validation loss directly to `Learner`.

```
# If registering this callback in construction
from functools import partial

learn = Learner(
    data, model,
    callback_fns=[partial(FastAIPruningCallback, trial=trial, monitor='valid_loss
↪')])
```

Example

Register a pruning callback to `learn.fit` and `learn.fit_one_cycle`.

```
learn.fit(n_epochs, callbacks=[FastAIPruningCallback(learn, trial, 'valid_loss')])
learn.fit_one_cycle(
    n_epochs, cyc_len, max_lr,
    callbacks=[FastAIPruningCallback(learn, trial, 'valid_loss')])
```

Parameters

- **learn** – fastai.basic_train.Learner.
- **trial** – A `Trial` corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g. `valid_loss` and `Accuracy`. Please refer to fastai.Callback reference for further details.

class ml_experiment.integrations.**MXNetPruningCallback**(*trial*, *eval_metric*)

MXNet callback to prune unpromising trials.

Example

Add a pruning callback which observes validation accuracy.

```
model.fit(train_data=X, eval_data=Y,
          eval_end_callback=MXNetPruningCallback(trial, eval_metric='accuracy'))
```

Parameters

- **trial** – A `Trial` corresponding to the current evaluation of the objective function.
- **eval_metric** – An evaluation metric name for pruning, e.g., `cross-entropy` and `accuracy`. If using default metrics like mxnet.metrics.Accuracy, use it's default metric name. For custom metrics, use the metric_name provided to constructor. Please refer to mxnet.metrics reference for further details.

class ml_experiment.integrations.**ChainerPruningExtension**(*trial*, *observation_key*, *pruner_trigger*)

Chainer extension to prune unpromising trials.

Example

Add a pruning extension which observes validation losses to Chainer Trainer.

```
trainer.extend(
    ChainerPruningExtension(trial, 'validation/main/loss', (1, 'epoch')))
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **observation_key** – An evaluation metric for pruning, e.g., main/loss and validation/main/accuracy. Please refer to chainer.Reporter reference for further details.
- **pruner_trigger** – A trigger to execute pruning. pruner_trigger is an instance of IntervalTrigger or ManualScheduleTrigger. IntervalTrigger can be specified by a tuple of the interval length and its unit like (1, 'epoch').

5.4 ml_experiment.config.models

class ml_experiment.config.models.**Metric**(***data*)

direction: OptimizationDirection = None

name: str = None

CHAPTER

SIX

CLI REFERENCE

ml-experiment CLI allows you to execute jobs from a configuration file. It can also be used to run a job without having to create a configuration file. And ultimately, it can be used to execute jobs combining input arguments with a configuration file, so those config files can work as a template.

Usage:

```
$ ml-experiment [OPTIONS] [SCRIPTS]...
```

Options:

- `--config_file FILE`
- `--name TEXT`: Name of the job. Overrides the config file field if specified.
- `--kind [job|experiment|group]`: Type of job. Overrides the config file field if specified.
- `--params FILE | DICT`: Job parameters. If config file is specified, these parameters In case of overlap, the values of this dictionary will take precedence over the rest
- `--param_space FILE | DICT`: Job parameter space. Only applies for groups of experiments. In case of overlap, the values of this dictionary will take precedence over the rest
- `--num_trials POSITIVE_INT`: Number of experiments to execute in parallel. Only applies for groups of experiments. Overrides the config file field if specified.
- `--timeout_per_trial POSITIVE_FLOAT`: Timeout per trial. In case of an experiment taking too long, it will be aborted.Only applies for groups of experiments. Overrides the config file field if specified.
- `--sampler [random|tpe|skopt]`: Sampler name. Only applies for groups of experiments. Overrides the config file field if specified.
- `--pruner [hyperband|sha|percentile|median]`: Pruner name. Only applies for groups of experiments. Overrides the config file field if specified.
- `--metric_key TEXT`: Name of the metric to optimize. It must be one of the keys of the metrics dictionary returned by the main function. Only applies for groups of experiments. Overrides the config file field if specified.
- `--metric_direction [minimize|maximize]`: Whether Hyperparameter Optimization Engine should minimize or maximize the given metric. Only applies for groups of experiments. Overrides the config file field if specified.
- `--docker_image TEXT`: If specified, the job will be run inside a docker contained based on the given image
- `--dockerfile FILE`
- `--docker_context DIRECTORY`: A directory to use as a docker context.Only applies when dockerfile is specified. Overrides the config file field if specified.

- `--docker_build_args FILE | DICT`: A dictionary of build arguments. Only applies when the Dockerfile is specified.
- `--ray_config FILE | DICT`: A dictionary of arguments to pass to Ray.init.Here you can specify the cluster address, number of cpu, gpu, etc. In case of overlap, the values of this dictionary will take precedence over the rest
- `--install-completion`: Install completion for the current shell.
- `--show-completion`: Show completion for the current shell, to copy it or customize the installation.
- `--help`: Show this message and exit.
- **User Guide:**