maggy Documentation

Release 0.4.0

Logical Clocks AB

Apr 19, 2020

Contents:

Maggy is a framework for efficient asynchronous optimization of expensive black-box functions on top of Apache Spark. Compared to existing frameworks, maggy is not bound to stage based optimization algorithms and therefore it is able to make extensive use of early stopping in order to achieve efficient resource utilization.

For a video describing Maggy, see this talk at the Spark/AI Summit.

Right now, maggy supports asynchronous hyperparameter tuning of machine learning and deep learning models, and ablation studies on neural network layers as well as input features.

Moreover, it provides a developer API that allows advanced usage by implementing custom optimization algorithms and early stopping criteria.

To accomodate asynchronous algorithms, support for communication between the Driver and Executors via RPCs through Maggy was added. The Optimizer that guides hyperparameter search is located on the Driver and it assigns trials to Executors. Executors periodically send back to the Driver the current performance of their trial, and the Optimizer can decide to early-stop any ongoing trial and send the Executor a new trial instead.

CHAPTER 1

Quick Start

To Install:

```
>>> pip install maggy
```

The programming model consists of wrapping the code containing the model training inside a function. Inside that wrapper function provide all imports and parts that make up your experiment.

There are three requirements for this wrapper function:

1. The function should take the hyperparameters as arguments, plus one additional parameter reporter which is needed for reporting the current metric to the experiment driver.
2. The function should return the metric that you want to optimize for. This should coincide with the metric being reported in the Keras callback (see next point).
3. In order to leverage on the early stopping capabilities of maggy, you need to make use of the maggy reporter API. By including the reporter in your training loop, you are telling maggy which metric to report back to the experiment driver for optimization and to check for global stopping. It is as easy as adding reporter.broadcast(metric=YOUR_METRIC) for example at the end of your epoch or batch training step and adding a reporter argument to your function signature. If you are not writing your own training loop you can use the pre-written Keras callbacks in the *maggy.callbacks* module.

Sample usage:

```
>>> # Define Searchspace
>>> from maggy import Searchspace
>>> # The searchspace can be instantiated with parameters
>>> sp = Searchspace(kernel=('INTEGER', [2, 8]), pool=('INTEGER', [2, 8]))
>>> # Or additional parameters can be added one by one
>>> sp.add('dropout', ('DOUBLE', [0.01, 0.99]))
```

```
>>> # Define training wrapper function:
>>> def mnist(kernel, pool, dropout, reporter):
>>>     # This is your training iteration loop
>>>     for i in range(number_iterations):
```

(continues on next page)

(continued from previous page)

```
>>>         ...
>>>         # add the maggy reporter to report the metric to be optimized
>>>         reporter.broadcast(metric=accuracy)
>>>         ...
>>>     # Return the same final metric
>>>     return accuracy
```

```
>>> # Launch maggy experiment
>>> from maggy import experiment
>>> result = experiment.lagom(map_fun=mnist,
>>>                           searchspace=sp,
>>>                           optimizer='randomsearch',
>>>                           direction='max',
>>>                           num_trials=15,
>>>                           name='MNIST'
>>>                          )
```

lagom is a Swedish word meaning "just the right amount". This is how maggy uses your resources.

CHAPTER 2

MNIST Example

For a full MNIST example with random search using Keras, see the Jupyter Notebook in the *examples* folder.

CHAPTER 3

Documentation

Read our blog post for more details.

API documentation is available here.

3.1 Maggy User API

3.1.1 maggy.experiment module

Experiment module used for running asynchronous optimization tasks.

The programming model is that you wrap the code containing the model training inside a wrapper function. Inside that wrapper function provide all imports and parts that make up your experiment, see examples below. Whenever a function to run an experiment is invoked it is also registered in the Experiments service along with the provided information.

maggy.experiment.**lagom**(*map_fun*, *name='no-name'*, *experiment_type='optimization'*, *searchspace=None*, *optimizer=None*, *direction='max'*, *num_trials=1*, *ablation_study=None*, *ablator=None*, *optimization_key='metric'*, *hb_interval=1*, *es_policy='median'*, *es_interval=300*, *es_min=10*, *description=''*)

Launches a maggy experiment, which depending on *experiment_type* can either be a hyperparameter optimization or an ablation study experiment. Given a search space, objective and a model training procedure *map_fun* (black-box function), an experiment is the whole process of finding the best hyperparameter combination in the search space, optimizing the black-box function. Currently maggy supports random search and a median stopping rule.

lagom is a Swedish word meaning "just the right amount".

Parameters

- **map_fun** (*function*) – User defined experiment containing the model training.
- **name** (*str*) – A user defined experiment identifier.

- **experiment_type** (*str*) – Type of Maggy experiment, either 'optimization' (default) or 'ablation'.
- **searchspace** (Searchspace) – A maggy Searchspace object from which samples are drawn.
- **optimizer** (*str, AbstractOptimizer*) – The optimizer is the part generating new trials.
- **direction** (*str*) – If set to 'max' the highest value returned will correspond to the best solution, if set to 'min' the opposite is true.
- **num_trials** (*int*) – the number of trials to evaluate given the search space, each containing a different hyperparameter combination
- **ablation_study** (AblationStudy) – Ablation study object. Can be None for optimization experiment type.
- **ablator** (*str, AbstractAblator*) – Ablator to use for experiment type 'ablation'.
- **optimization_key** (*str, optional*) – Name of the metric to be optimized
- **hb_interval** (*int, optional*) – The heartbeat interval in seconds from trial executor to experiment driver, defaults to 1
- **es_policy** (*str, optional*) – The earlystopping policy, defaults to 'median'
- **es_interval** (*int, optional*) – Frequency interval in seconds to check currently running trials for early stopping, defaults to 300
- **es_min** (*int, optional*) – Minimum number of trials finalized before checking for early stopping, defaults to 10
- **description** (*str, optional*) – A longer description of the experiment.

Raises **RuntimeError** – An experiment is currently running.

Returns A dictionary indicating the best trial and best hyperparameter combination with it's performance metric

Return type dict

3.1.2 maggy.searchspace module

class maggy.**Searchspace**(***kwargs*)

Create an instance of *Searchspace* from keyword arguments.

A searchspace is essentially a set of key value pairs, defining the hyperparameters with a name, type and a feasible interval. The keyword arguments specify name-values pairs for the hyperparameters, where values are tuples of the form (type, list). Type is a string with one of the following values:

- DOUBLE
- INTEGER
- DISCRETE
- CATEGORICAL

And the list in the tuple specifies either two values only, the start and end point of of the feasible interval for DOUBLE and INTEGER, or the discrete possible values for the types DISCRETE and CATEGORICAL.

Sample usage:

```
>>> # Define Searchspace
>>> from maggy import Searchspace
>>> # The searchspace can be instantiated with parameters
>>> sp = Searchspace(kernel=('INTEGER', [2, 8]), pool=('INTEGER', [2, 8]))
>>> # Or additional parameters can be added one by one
>>> sp.add('dropout', ('DOUBLE', [0.01, 0.99]))
```

The *Searchspace* object can also be initialized from a python dictionary:

```
>>> sp_dict = sp.to_dict()
>>> sp_new = Searchspace(**sp_dict)
```

The parameter names are added as attributes of *Searchspace* object, so they can be accessed directly with the dot notation *searchspace._name_*.

add(*name*, *value*)
Adds {name, value} pair to hyperparameters.

Parameters

- **name** (*str*) – Name of the hyperparameter
- **value** (*tuple*) – A tuple of the parameter type and its feasible region

Raises

- **ValueError** – Hyperparameter name is reserved
- **ValueError** – Hyperparameter feasible region in wrong format

get(*name*, *default=None*)
Returns the value of *name* if it exists, else *default*.

get_random_parameter_values(*num*)
Generate random parameter dictionaries, e.g. to be used for initializing an optimizer.

Parameters **num** (*int*) – number of random parameter dictionaries to be generated.

Raises **ValueError** – *num* is not an int.

Returns a list containing parameter dictionaries

Return type list

items()
Returns a sorted iterable over all hyperparameters in the searchspace.

Allows to iterate over the hyperparameters in a searchspace. The parameters are sorted in the order of which they were added to the searchspace by the user.

Returns an iterable of the searchspace

Type *Searchspace*

keys()
Returns a sorted iterable list over the names of hyperparameters in the searchspace.

Returns names of hyperparameters as a list of strings

Type list

names()
Returns the dictionary with the names and types of all hyperparameters.

Returns Dictionary of hyperparameter names, with types as value

Return type dict

to_dict()
Return the hyperparameters as a Python dictionary.

Returns A dictionary with hyperparameter names as keys. The values are the hyperparameter values.

Return type dict

values()
Returns a sorted iterable list over the types and feasible intervals of hyperparameters in the searchspace.

Returns types and feasible interval of hyperparameters as tuple

Type tuple

3.1.3 maggy.callbacks module

class maggy.callbacks.**KerasBatchEnd**(*reporter*, *metric='loss'*)
A Keras callback reporting a specified *metric* at the end of the batch to the maggy experiment driver.

loss is always available as a metric, and optionally *acc* (if accuracy monitoring is enabled, that is, accuracy is added to keras model metrics). Validation metrics are not available for the BatchEnd callback. Validation after every batch would be too expensive. Default is training loss (*loss*).

Example usage:

```
>>> from maggy.callbacks import KerasBatchEnd
>>> callbacks = [KerasBatchEnd(reporter, metric='acc')]
```

class maggy.callbacks.**KerasEpochEnd**(*reporter*, *metric='val_loss'*)
A Keras callback reporting a specified *metric* at the end of an epoch to the maggy experiment driver.

val_loss is always available as a metric, and optionally *val_acc* (if accuracy monitoring is enabled, that is, accuracy is added to keras model metrics). Training metrics are available under the names *loss* and *acc*. Default is validation loss (*val_loss*).

Example usage:

```
>>> from maggy.callbacks import KerasBatchEnd
>>> callbacks = [KerasBatchEnd(reporter, metric='val_acc')]
```

3.1.4 maggy.ablation module

class maggy.ablation.**AblationStudy**(*training_dataset_name*, *training_dataset_version*, *label_name*, ***kwargs*)
The *AblationStudy* object is the entry point to define an ablation study with maggy. This object can subsequently be passed as an argument when the experiment is launched with *experiment.lagom()*.

Sample usage:

```
>>> from maggy.ablation import AblationStudy
>>> ablation_study = AblationStudy('titanic_train_dataset',
>>>     label_name='survived')
```

The above code will create an *AblationStudy* instance with a default dataset generator function, which uses the project feature store to return a *TFRecordDataset* based on the feature ablation configuration (for an example, look at *ablator.LOCO.get_dataset_generator()*). If you want to provide your own dataset generator function,

define it before creating the *AblationStudy* instance and pass it to the initializer. In the example below we assume the user has created a function called *create_tf_dataset()* that returns a *TFRecordDataset*:

```
>>> ablation_study = AblationStudy('titanic_train_dataset',
        label_name='survived', dataset_generator=create_tf_dataset)
```

In case you want to perform feature ablation with your custom dataset generator function, then of course your function should be able to return specific datasets based on the feature ablation configuration. For an example implementation of such logic, look at *ablator.LOCO.get_dataset_generator()*.

After creating your *AblationStudy* instance, you should define your study configuration by including layers and features that you want to be ablated:

```
>>> ablation_study.features.include('pclass', 'fare')
>>> ablation_study.model.layers.include('my_dense_two',
>>>     'my_dense_three')
```

You can also add a layer group using a list:

```
>>> ablation_study.model.layers.include_groups(['my_dense_two',
>>>     'my_dense_four'])
```

Or add a layer group using a prefix:

```
>>> ablation_study.model.layers.include_groups(prefix='my_dense')
```

Next you should define a base model function using the layer and feature names you previously specified:

```
>>> # you only need to add the `name` parameter to layer initializers
>>> def base_model_generator():
>>>     model = tf.keras.Sequential()
>>>     model.add(tf.keras.layers.Dense(64, activation='relu'))
>>>     model.add(tf.keras.layers.Dense(..., name='my_dense_two', ...)
>>>     model.add(tf.keras.layers.Dense(32, activation='relu'))
>>>     model.add(tf.keras.layers.Dense(..., name='my_dense_sigmoid', ...)
>>>     # output layer
>>>     model.add(tf.keras.layers.Dense(1, activation='linear'))
>>>     return model
```

Make sure to include the generator function in the study:

```
>>> ablation_study.model.set_base_model_generator(base_model_generator)
```

Last but not least you can define your actual training function:

```
>>> from maggy import experiment

>>> def training_function(dataset_function, model_function):
>>>     import tensorflow as tf
>>>     epochs = 5
>>>     batch_size = 10
>>>     tf_dataset = dataset_function(epochs, batch_size)
>>>     model = model_function()
>>>     model.compile(optimizer=tf.train.AdamOptimizer(0.001),
>>>             loss='binary_crossentropy',
>>>             metrics=['accuracy'])
```

```
>>>     history = model.fit(tf_dataset, epochs=epochs, steps_per_epoch=30)
>>>     return float(history.history['acc'][-1])
```

Lagom the experiment:

```
>>> result = experiment.lagom(map_fun=training_function,
>>>                           experiment_type='ablation',
>>>                           ablation_study=ablation_study,
>>>                           ablator='loco',
>>>                           name='Titanic-LOCO')
```

__init__(*training_dataset_name*, *training_dataset_version*, *label_name*, ***kwargs*)
Initializes the ablation study.

Parameters

- **training_dataset_name** (*str*) – Name of the training dataset in the featurestore.
- **training_dataset_version** (*int*) – Version of the training dataset to be used.
- **label_name** (*str*) – Name of the target prediction label.

to_dict()
Returns the ablation study configuration as a Python dictionary.

Returns A dictionary with ablation study configuration parameters as keys (i.e. 'training_dataset_name', 'included_features', etc.)

Return type dict

3.2 Maggy Developer API

As a developer you have the possibility to implement your custom optimizers or ablators. For that you can implement an abstract method, which you can then pass as an argument when launching the experiment. For examples, please look at existing optimizers and ablators.

3.2.1 maggy.optimizer module

3.2.2 maggy.ablation.ablator module

class maggy.ablation.ablator.abstractablator.**AbstractAblator**(*ablation_study*, *final_store*)

finalize_experiment(*trials*)
This method will be called before finishing the experiment. Developers can implement this method e.g. for cleanup or extra logging.

get_dataset_generator(*ablated_feature*, *dataset_type='tfrecord'*)
Create and return a dataset generator function based on the ablation policy to be used in a trial. The returned function will be executed on the executor per each trial.

Parameters

- **ablated_feature** (*str*) – the name of the feature to be excluded from the training dataset. Must match a feature name in the corresponding feature group in the feature store.
- **dataset_type** – type of the dataset. For now, we only support 'tfrecord'.

Returns A function that generates a TFRecordDataset

Return type function

get_number_of_trials()
If applicable, calculate and return the total number of trials of the ablation experiment. Make sure to also include the base (reference) trial in the count.

Returns total number of trials of the ablation study experiment

Return type int

get_trial(*ablation_trial=None*)
Return a *Trial* to be assigned to an executor, or *None* if there are no trials remaining in the experiment. The trial should contain a dataset generator and a model generator. Depending on the ablator policy, the trials could come from a list (buffer) of pre-made trials, or generated on the fly.

Return type *Trial* or *None*

initialize()
Initialize the ablation study experiment by generating a number of trials. Depending on the ablation policy, this method might generate all the trials (e.g. as in LOCO), or generate a number of trials to warm-start the experiment. The trials should be added to *trial_buffer* in form of *Trial* objects.

3.3 Release 0.1

3.4 License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

F

G

I

K

L

M

N

S

T

V