redisai-py

Release 1.0.2

May 19, 2021

Contents:

CHAPTER 1

redisai-py

redisai-py is the Python client for RedisAI. Checkout the documentation for API details and examples

1.1 Installation

1. Install Redis 5.0 or above
2. Install RedisAI
3. Install the Python client

```
$ pip install redisai
```

4. Install serialization-deserialization utility (optional)

```
$ pip install ml2rt
```

RedisAI example repo shows few examples made using redisai-py under *python_client* folder. Also, checkout ml2rt for convenient functions those might help in converting models (sparkml, sklearn, xgboost to ONNX), serializing models to disk, loading it back to redisai-py etc.

CHAPTER 2

API Documentation

This page hosts the documentation of all user-facing APIs. The only entrypoint the users should be using is the `Client` class. It exposes all the RedisAI commands as class methods

2.1 Client Class

class Client(*debug=False*, *enable_postprocess=True*, **args*, ***kwargs*)

Redis client build specifically for the RedisAI module. It takes all the necessary parameters to establish the connection and an optional `debug` parameter on initialization

Parameters

- **debug** (*bool*) – If debug mode is ON, then each command that is sent to the server is printed to the terminal
- **enable_postprocess** (*bool*) – Flag to enable post processing. If enabled, all the bytestring-ed returns are converted to python strings recursively and key value pairs will be converted to dictionaries. Also note that, this flag doesn't work with pipeline() function since pipeline function could have native redis commands (along with RedisAI commands)

Example

```
>>> from redisai import Client
>>> con = Client(host='localhost', port=6379)
```

dag(*load: Sequence[T_co] = None*, *persist: Sequence[T_co] = None*, *readonly: bool = False*) → redisai.dag.Dag

It returns a DAG object on which other DAG-allowed operations can be called. For more details about DAG in RedisAI, refer to the RedisAI documentation.

Parameters

- **load** (*Union[AnyStr, List[AnyStr]]*) – Load the list of given values from the keyspace to DAG scope
- **persist** (*Union[AnyStr, List[AnyStr]]*) – Write the list of given key, values to the keyspace from DAG scope
- **readonly** (*bool*) – If True, it triggers AI.DAGRUN_RO, the read only DAG which cannot write (PERSIST) to the keyspace. But since it can't write, it can execute on replicas

Returns Dag object which holds other operations permitted inside DAG as attributes

Return type Any

Example

```
>>> con.tensorset('tensor', ...)
'OK'
>>> con.modelstore('model', ...)
'OK'
>>> dag = con.dag(load=['tensor'], persist=['output'])
>>> dag.tensorset('another', ...)
>>> dag.modelrun('model', inputs=['tensor', 'another'], outputs=['output'])
>>> output = dag.tensorget('output').run()
>>> # You can even chain the operations
>>> result = dag.tensorset(**akwargs).modelrun(**bkwargs).
↪tensorget(**ckwargs).run()
```

infoget (*key: AnyStr*) → dict

Get information such as - How long since the model has been running - How many samples have been processed - How many calls handled - How many errors raised - etc.

Parameters **key** (*AnyStr*) – Model key

Returns Dictionary of model run details

Return type dict

Example

```
>>> con.infoget('m')
{'key': 'm', 'type': 'MODEL', 'backend': 'TF', 'device': 'cpu', 'tag': '',
'duration': 0, 'samples': 0, 'calls': 0, 'errors': 0}
```

inforeset (*key: AnyStr*) → str

Reset the run information about the model

Parameters **key** (*AnyStr*) – Model key

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.inforeset('m')
'OK'
```

loadbackend (*identifier: AnyStr*, *path: AnyStr*) → str

RedisAI by default won't load any backends. User can either explicitly load the backend by using this function or let RedisAI load the required backend from the default path on-demand.

Parameters

- **identifier** (*str*) – Representing which backend. Allowed values - TF, TFLITE, TORCH & ONNX
- **path** (*str*) – Path to the shared object of the backend

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.loadbackend('TORCH', '/path/to/the/backend/redisai_torch.so')
'OK'
```

modeldel (*key: AnyStr*) → str

Delete the model from the RedisAI server

Parameters **key** (*AnyStr*) – Key of the model to be deleted

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.modeldel('model')
'OK'
```

modelexecute (*key: AnyStr, inputs: Union[AnyStr, List[AnyStr]], outputs: Union[AnyStr, List[AnyStr]], timeout: int = None*) → str

Run the model using input(s) which are already in the scope and are associated to some keys. Modelexecute also needs the output key name(s) to store the output from the model. The number of outputs from the model and the number of keys provided here must be same. Otherwise, RedisAI throws an error

Parameters

- **key** (*str*) – Model key to run
- **inputs** (*Union[AnyStr, List[AnyStr]]*) – Tensor(s) which is already saved in the RedisAI using a tensorset call. These tensors will be used as the inputs for the modelexecute
- **outputs** (*Union[AnyStr, List[AnyStr]]*) – keys on which the outputs to be saved. If those keys exist already, modelexecute will overwrite them with new values
- **timeout** (*int*) – The max number on milisecinds that may pass before the request is prossced (meaning that the result will not be computed after that time and TIMEDOUT is returned in that case

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.modelstore('m', 'tf', 'cpu', model_pb,
...                inputs=['a', 'b'], outputs=['mul'], tag='v1.0')
'OK'
>>> con.tensorset('a', (2, 3), dtype='float')
'OK'
>>> con.tensorset('b', (2, 3), dtype='float')
'OK'
>>> con.modelexecute('m', ['a', 'b'], ['c'])
'OK'
```

modelget (*key: AnyStr*, *meta_only=False*) → dict

Fetch the model details and the model blob back from RedisAI

Parameters

- **key** (*AnyStr*) – Model key in RedisAI
- **meta_only** (*bool*) – If True, only the meta data will be fetched, not the model blob

Returns A dictionary of model details such as device, backend etc. The model blob will be available at the key 'blob'

Return type dict

Example

```
>>> con.modelget('model', meta_only=True)
{'backend': 'TF', 'device': 'cpu', 'tag': 'v1.0'}
```

modelrun (*key: AnyStr, inputs: Union[AnyStr, List[AnyStr]], outputs: Union[AnyStr, List[AnyStr]]*) → str

Run the model using input(s) which are already in the scope and are associated to some keys. Modelrun also needs the output key name(s) to store the output from the model. The number of outputs from the model and the number of keys provided here must be same. Otherwise, RedisAI throws an error

Parameters

- **key** (*str*) – Model key to run
- **inputs** (*Union[AnyStr, List[AnyStr]]*) – Tensor(s) which is already saved in the RedisAI using a tensorset call. These tensors will be used as the input for the modelrun
- **outputs** (*Union[AnyStr, List[AnyStr]]*) – keys on which the outputs to be saved. If those keys exist already, modelrun will overwrite them with new values

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.modelstore('m', 'tf', 'cpu', model_pb,
...                inputs=['a', 'b'], outputs=['mul'], tag='v1.0')
'OK'
>>> con.tensorset('a', (2, 3), dtype='float')
'OK'
```

(continues on next page)

(continued from previous page)

```
>>> con.tensorset('b', (2, 3), dtype='float')
'OK'
>>> con.modelrun('m', ['a', 'b'], ['c'])
'OK'
```

modelscan() → List[List[AnyStr]]

Returns the list of all the models in the RedisAI server. Modelscan API is currently experimental and might be removed or changed in the future without warning

Returns List of list of models and tags for each model if they existed

Return type List[List[AnyStr]]

Example

```
>>> con.modelscan()
[['pt_model', ''], ['m', 'v1.2']]
```

modelset(*key: AnyStr*, *backend: str*, *device: str*, *data: ByteString*, *batch: int = None*, *minbatch: int = None*, *tag: AnyStr = None*, *inputs: Union[AnyStr, List[AnyStr]] = None*, *outputs: Union[AnyStr, List[AnyStr]] = None*) → str

Set the model on provided key.

Parameters

- **key** (*AnyStr*) – Key name
- **backend** (*str*) – Backend name. Allowed backends are TF, TORCH, TFLITE, ONNX
- **device** (*str*) – Device name. Allowed devices are CPU and GPU. If multiple GPUs are available, it can be specified using the format GPU:<gpu number>. For example: GPU:0
- **data** (*bytes*) – Model graph read as bytes string
- **batch** (*int*) – Number of batches for doing auto-batching
- **minbatch** (*int*) – Minimum number of samples required in a batch for model execution
- **tag** (*AnyStr*) – Any string that will be saved in RedisAI as tag for the model
- **inputs** (*Union[AnyStr, List[AnyStr]]*) – Input node(s) in the graph. Required only Tensorflow graphs
- **outputs** (*Union[AnyStr, List[AnyStr]]*) – Output node(s) in the graph Required only for Tensorflow graphs

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> # Torch model
>>> model_path = os.path.join('path/to/TorchScriptModel.pt')
>>> model = open(model_path, 'rb').read()
>>> con.modelset("model", 'torch', 'cpu', model, tag='v1.0')
'OK'
>>> # Tensorflow model
```

(continues on next page)

(continued from previous page)

```
>>> model_path = os.path.join('/path/to/tf_frozen_graph.pb')
>>> model = open(model_path, 'rb').read()
>>> con.modelset('m', 'tf', 'cpu', model,
...              inputs=['a', 'b'], outputs=['mul'], tag='v1.0')
'OK'
```

modelstore(*key: AnyStr*, *backend: str*, *device: str*, *data: ByteString*, *batch: int = None*, *minbatch: int = None*, *minbatchtimeout: int = None*, *tag: AnyStr = None*, *inputs: Union[AnyStr, List[AnyStr]] = None*, *outputs: Union[AnyStr, List[AnyStr]] = None*) → str

Set the model on provided key.

Parameters

- **key** (*AnyStr*) – Key name
- **backend** (*str*) – Backend name. Allowed backends are TF, TORCH, TFLITE, ONNX
- **device** (*str*) – Device name. Allowed devices are CPU and GPU. If multiple GPUs are available, it can be specified using the format GPU:<gpu number>. For example: GPU:0
- **data** (*bytes*) – Model graph read as bytes string
- **batch** (*int*) – Number of batches for doing auto-batching
- **minbatch** (*int*) – Minimum number of samples required in a batch for model execution
- **minbatchtimeout** (*int*) – The max number of miliseconds for which the engine will not trigger an execution if the number of samples is lower than minbatch (after minbatchtimeout is passed, the execution will start even if minbatch jas not reached)
- **tag** (*AnyStr*) – Any string that will be saved in RedisAI as tag for the model
- **inputs** (*Union[AnyStr, List[AnyStr]]*) – Input node(s) in the graph. Required only Tensorflow graphs
- **outputs** (*Union[AnyStr, List[AnyStr]]*) – Output node(s) in the graph Required only for Tensorflow graphs

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> # Torch model
>>> model_path = os.path.join('path/to/TorchScriptModel.pt')
>>> model = open(model_path, 'rb').read()
>>> con.modeltore("model", 'torch', 'cpu', model, tag='v1.0')
'OK'
>>> # Tensorflow model
>>> model_path = os.path.join('/path/to/tf_frozen_graph.pb')
>>> model = open(model_path, 'rb').read()
>>> con.modelstore('m', 'tf', 'cpu', model,
...              inputs=['a', 'b'], outputs=['mul'], tag='v1.0')
'OK'
```

pipeline(*transaction: bool = True*, *shard_hint: bool = None*) → redisai.pipeline.Pipeline

It follows the same pipeline implementation of native redis client but enables it to access redisai operation as well. This function is experimental in the current release.

Example

```
>>> pipe = con.pipeline(transaction=False)
>>> pipe = pipe.set('nativeKey', 1)
>>> pipe = pipe.tensorset('redisaiKey', np.array([1, 2]))
>>> pipe.execute()
[True, b'OK']
```

scriptdel (*key: AnyStr*) → str
Delete the script from the RedisAI server

Parameters **key** (*AnyStr*) – Script key to be deleted

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.scriptdel('ket')
'OK'
```

scriptget (*key: AnyStr*, *meta_only=False*) → dict
Get the saved script from RedisAI. Operation similar to model get

Parameters

- **key** (*AnyStr*) – Key of the script
- **meta_only** (*bool*) – If True, only the meta data will be fetched, not the script itself

Returns Dictionary of script details which includes the script at the key `source`

Return type dict

Example

```
>>> con.scriptget('ket', meta_only=True)
{'device': 'cpu'}
```

scriptrun (*key: AnyStr, function: AnyStr, inputs: Union[AnyStr, Sequence[AnyStr]], outputs: Union[AnyStr, Sequence[AnyStr]]*) → str
Run an already set script. Similar to modelrun

Parameters

- **key** (*AnyStr*) – Script key
- **function** (*AnyStr*) – Name of the function in the `script`
- **inputs** (*Union[AnyStr, List[AnyStr]]*) – Tensor(s) which is already saved in the RedisAI using a tensorset call. These tensors will be used as the input for the modelrun
- **outputs** (*Union[AnyStr, List[AnyStr]]*) – keys on which the outputs to be saved. If those keys exist already, modelrun will overwrite them with new values

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.scriptrun('ket', 'bar', inputs=['a', 'b'], outputs=['c'])
'OK'
```

scriptscan() → List[List[AnyStr]]

Returns the list of all the script in the RedisAI server. Scriptscan API is currently experimental and might remove or change in the future without warning

Returns List of list of scripts and tags for each script if they existed

Return type List[List[AnyStr]]

Example

```
>>> con.scriptscan()
[['ket1', 'v1.0'], ['ket2', '']]
```

scriptset(*key: AnyStr*, *device: str*, *script: str*, *tag: AnyStr = None*) → str

Set the script to RedisAI. Action similar to Modelset. RedisAI uses the TorchScript engine to execute the script. So the script should have only TorchScript supported constructs. That being said, it's important to mention that using redisai script to do post processing or pre processing for a Tensorflow (or any other backend) is completely valid. For more details about TorchScript and supported ops, checkout TorchScript documentation.

Parameters

- **key** (*AnyStr*) – Script key at the server
- **device** (*str*) – Device name. Allowed devices are CPU and GPU. If multiple GPUs are available. it can be specified using the format GPU:<gpu number>. For example: GPU:0
- **script** (*str*) – Script itself, as a Python string
- **tag** (*AnyStr*) – Any string that will be saved in RedisAI as tag for the model

Returns 'OK' if success, raise an exception otherwise

Return type str

Note: Even though `script` is pure Python code, it's a subset of Python language and not all the Python operations are supported. For more details, checkout TorchScript documentation. It's also important to note that that the script is executed on a high performance C++ runtime instead of the Python interpreter. And hence `script` should not have any import statements (A common mistake people make all the time)

Example

```
>>> script = open(scriptpath).read()
>>> con.scriptset('ket', 'cpu', script)
'OK'
```

tensorget(*key: AnyStr*, *as_numpy: bool = True*, *as_numpy_mutable: bool = False*, *meta_only: bool = False*) → Union[dict, numpy.ndarray]

Retrieve the value of a tensor from the server. By default it returns the numpy array but it can be controlled using the *as_type* and *meta_only* argument.

Parameters

- **key** (*AnyStr*) – The name of the tensor
- **as_numpy** (*bool*) – If True, returns a numpy.ndarray. Returns the value as a list and the metadata in a dictionary if False. This flag also decides how to fetch the value from the RedisAI server, which also has performance implications
- **as_numpy_mutable** (*bool*) – If True, returns a a mutable numpy.ndarray object by copy the tensor data. Otherwise (as long as_numpy=True) the returned numpy.ndarray will use the original tensor buffer and will be for read-only
- **meta_only** (*bool*) – If True, the value is not retrieved, only the shape and the type

Returns Returns a dictionary of data or a numpy array. Default is numpy array

Return type Union[dict, np.ndarray]

Example

```
>>> con.tensorget('x')
array([2, 3, 4])
>>> con.tensorget('x' as_numpy=False)
{'values': [2, 3, 4], 'dtype': 'INT64', 'shape': [3]}
>>> con.tensorget('x', meta_only=True)
{'dtype': 'INT64', 'shape': [3]}
```

tensorset(*key: AnyStr, tensor: Union[numpy.ndarray, list, tuple], shape: Sequence[int] = None, dtype: str = None*) → str

Set the tensor to a key in RedisAI

Parameters

- **key** (*AnyStr*) – The name of the tensor
- **tensor** (*Union[np.ndarray, list, tuple]*) – A *np.ndarray* object or Python list or tuple
- **shape** (*Sequence[int]*) – Shape of the tensor. Required if *tensor* is list or tuple
- **dtype** (*str*) – Data type of the tensor. Required if *tensor* is list or tuple

Returns 'OK' if success, raise an exception otherwise

Return type str

Example

```
>>> con.tensorset('a', (2, 3), dtype='float')
'OK'
>>> input_array = np.array([2, 3], dtype=np.float32)
>>> con.tensorset('x', input_array)
'OK'
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index